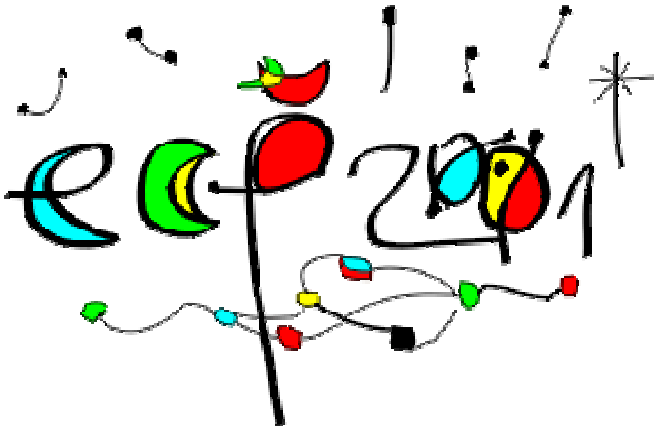
Pre-proceedings of the
Sixth European Conference on Planning



ECP-01
Toledo, Spain
September 12-14, 2001

Edited by

Amedeo Cesta and **Daniel Borrajo**

Pre-proceedings of the Sixth European Conference on Planning

ECP-01

Museum Victorio Macho
Toledo, Spain
September 12-14, 2001

Edited by

Amedeo Cesta and **Daniel Borrajo**

Editors

Amedeo Cesta
National Research Council of Italy
IP-CNR [PST]
Viale Marx, 15
I-00137 Rome, Italy
e-mail: *cesta@ip.rm.cnr.it*

Daniel Borrajo
Departamento de Informatica - ScaLAB
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911-Leganes, Madrid, Spain
e-mail: *dborrajo@ia.uc3m.es*

These Pre-proceedigs are printed with a grant from IP-CNR, Institute of Psychology of the Italian National Research Council, Rome, Italy.

ECP is a major international conference for presentation of new research in AI Planning and Scheduling, and a fruitful opportunity for contact and cross-fertilization among the different "souls" in the field. It has taken place in Europe every other year since 1991. It has evolved very quickly from a restricted workshop mainly devoted to the presentation of European research to a well established conference devoted to the presentation of rigorous and innovative research results from the international community. The sixth ECP conference takes place in the center of historical Toledo, the very well known old Spanish city, crossing of many different cultures. ECP-01 will follow its established scientific tradition, including events that highlight specific aspects of planning and scheduling research in the new millennium.

The Conference program includes long, short and demo paper presentations, an open Benchmark, Poster and Demo Session and three invited talks given by Prof. Hector Geffner, Prof. Vladimir Lifschitz and Prof. Pascal Van Hentenryck.

Papers in these proceedings are the result of a quite severe selection made by the Program Committee members out of the 92 submitted papers. We acknowledge the support of a number of additional reviewers that with their expertise helped in the selection process.

Program Committee

Ruth Aylett (University of Salford)
Chris Beck (ILOG S.A.)
Michael Beetz (University of Munich)
Susanne Biundo (University of Ulm)
Daniel Borrajo (Universidad Carlos III de Madrid -- local chair)
Luis Castillo (Universidad de Granada)
Amedeo Cesta (Nat. Research Council of Italy -- programme chair)
Steve Chien (Jet Propulsion Laboratory)
Berthe Choueiry (University of Nebrasca at Lincoln)
Rina Dechter (University of California at Irvine)
Giuseppe De Giacomo (University of Rome “La Sapienza”)
Maria Fox (University of Durham)
Hector Geffner (Universidad Simon Bolivar)
Alfonso Gerevini (University of Brescia)
Malik Ghallab (LAAS-CNRS, Toulouse)
Enrico Giunchiglia (University of Genoa)
Joachim Hertzberg (GMD, St.Augustin)
Peter Jonsson (University of Linkoping)
Subbarao Kambhampati (Arizona State University)
Jana Koehler (IBM Research, Zurich)
Sven Koenig (Georgia Institute of Technology)
Claude Le Pape (ILOG S.A.)
Lee McCluskey (University of Huddersfield)
Alfredo Milani (University of Perugia)
Nicola Muscettola (NASA Ames Research Center)
Karen Myers (SRI, Menlo Park)
Martha Pollack (University of Michigan)
Jussi Rintanen (Albert-Ludwigs-University Freiburg)
Alessandro Saffiotti (University of Orebro)
Camilla Schwind (LIM-CNRS, Marseille)
David E. Smith (NASA Ames Research Center)
Stephen F. Smith (Carnegie Mellon University)
Sam Steel (University of Essex)
Sylvie Thiebaux (CSIRO, Canberra)
Paolo Traverso (IRST, Trento)
Manuela Veloso (Carnegie Mellon University)

Additional Reviewers

Riccardo Aler

Jeremy Baxter
Ralph Becket
Piergiorgio Bertoli
Guido Boella
Ronen Brafman
Brett Browing

Alessandro Cimatti

Binh Minh Do

Nader Faisal Mohamed
Jeremy Frank
David Furcy

Max Garagnani
Ian Gent
Antonio Gonzales
Tim Grant
Emanuel Guere

Patrick Haslum
Laurie Hiyakumoto
Joerg Hoffmann

Froduald Kabanza
Gal A. Kaminka
Lars Karlsson
Gerry Kelleher

Rune M. Jensen

Peter Jarvis

Luca Iocchi

Philippe Laborie
John Levine
Yaxin Liu

Angelo Oddi

Andrew J. Parkes
Marco Pistore
Patrick Prosser

Riccardo Rosati
Marco Roveri

Romeo Sanchez Nigenda
Araceli Sanchis
Roberto Sebastiani
Ivan Serina
Francis Sourd
Biplav Srivastava

Gérard Verfaillie
Thierry Vidal
Vincent Vidal

Romain Trinquart
Ioannis Tsamardinos

Terry Zimmerman

ECP-01 Sponsors

Artificial Intelligence Journal

PLANET 2, the Network of Excellence in AI Planning

Ministerio de Ciencia y Tecnología (MCYT)

APSOLVE, a BTexaCT Technologies Company

AI*IA, the Italian Association for Artificial Intelligence

IP-CNR, Institute of Psychology of the Italian National Research Council

Universidad Carlos III de Madrid

ScALAB, Systems, Complex and Adaptive Laboratory

PST, Planning and Scheduling Team at IP-CNR

Table of Contents

Long Papers

September 12

September 13

September 14

Short Papers

Demo Papers

Long Papers

September 12

Combining Progression and Regression in State-Space Heuristic Planning

Dimitris Vrakas and Ioannis Vlahavas
Department of Informatics
Aristotle University of Thessaloniki
54006, GREECE
[dvrakas,vlahavas]@csd.auth.gr
Fax: ++3031 998419

Abstract. One of the most promising trends in Domain Independent AI Planning, nowadays, is state-space heuristic planning. The planners of this category construct general but efficient heuristic functions, which are used as a guide to traverse the state space either in a forward or a in backward direction. Although specific problems may favor one or the other direction, there is no clear evidence why any of them should be generally preferred.

This paper proposes a hybrid search strategy that combines search in both directions. The search begins from the *Initial State* in a forward direction and proceeds with a weighted A* search until no further improving states can be found. At that point, the algorithm changes direction and starts regressing the *Goals* trying to reach the best state found at the previous step. The direction of the search may change several times before a solution can be found. Two domain-independent heuristic functions based on ASP/HSP planners enhanced with a *Goal Ordering* technique have been implemented. The whole bi-directional planning system, named BP, was tested on a variety of problems adopted from the recent AIPS-00 planning competition with quite promising results. The paper also discusses the subject of domain analysis for state-space planning and proposes two methods for the elimination of redundant information from the problem definition and for the identification of independent sub-problems.

Keywords: Planning, Heuristic Search, Bi-Directional Search

1. Introduction

Motivated by the work of Drew McDermott in the mid 90's on heuristic state-space planning, a number of researchers turned to this direction. During the last few years a great amount of work has been done in the area of domain-independent, state-space, heuristic planning and a significant number of planning systems with remarkable performance were developed.

Hector Geffner in his recent work on HSP-2 [3] studies the matter of search direction and the HSP-2 planning system enables the user to decide for the direction of the search. It is clear from the experimental results that there are specific problems, which favor one or the other search directions, but in general there is no clear evidence why any of the two directions should be preferred.

In this paper we propose a hybrid search strategy for domain-independent, state-space heuristic planning that combines both progression (forward chaining) and regression (backward chaining). The search begins from the *Initial State* and proceeds with a weighted A* search until no further improving states can be found from the

Goals. At that point the algorithm changes direction and regress the *Goals* trying to reach the best state found at the previous step. The direction of the search may change several times before a solution can be found.

Two domain-independent heuristic functions based on ASP/HSP enhanced with a *Goal Ordering* technique were implemented and the whole bi-directional planning system, named BP, was tested on a variety of problems adopted from the recent AIPS-00 planning competition with quite promising results.

This paper also discusses the subject of automatic domain analysis and the utilization of the information extracted from it in the planning process. We propose two methods, based on the planning graphs created by the heuristic functions of BP, which can identify valuable information about the internal structure of the problems. The methods are utilized in BP for eliminating redundant information from the definition of the domain and for dividing the problem in a number of easier sub-problems that can be tackled in parallel.

The rest of the paper is organized as follows: Section 2 provides a brief review of the work related to state-space, heuristic planning and other approaches to bi-directional planning. Section 3 describes the bi-directional search strategy in detail and deals with certain issues that arise while regressing the goals of a problem. Section 4 describes the heuristic functions of BP and describes the adoption of a Goal Ordering technique to heuristic state space planning. Section 5 presents experimental results that illustrate the efficiency of BP on a variety of problems adopted from the AIPS-00 planning competition. Section 6 discusses a number of domain analysis techniques that result from the construction of planning graphs and section 7 concludes the paper and poses future directions.

2. Related Work

One of the most promising trends in domain-independent planning was presented over the last few years. It is based on a relatively simple idea where a general domain independent heuristic function is embodied in a heuristic search algorithm such as Hill Climbing, Best-First Search or A*. A detailed survey of search algorithms can be found in [9]. Examples of planning systems in this category are UNPOP[13], the ASP/HSP family [2,3,4], GRT[16], AltAlt[14] and FF[7].

The first planner in this category was UNPOP[13], a regression planner that constructed at each step a graph, named *greedy regression-match graph*. The graph was used in the search, for creating the heuristic function and cutting down the branching factor by pruning certain actions.

The direct descendant of UNPOP was HSP[4], which searches the state space in the forward direction though and constructs a more sophisticated heuristic function from a similar graph. HSP was followed by HSP-R[2], a similar planer with two main differences from HSP. The search is performed in a backwards manner and the heuristic is created in the opposite direction, which enables HSP-R to create the planning graph only once. HSP and HSP-R were later embodied in a unifying planning system called HSP-2[3]. GRT [18] is another extension to HSP that searches in the forward direction and creates the heuristic backwards once at the beginning.

Nigenda, Nguyen and Kambhampati presented a hybrid planning system, named AltAlt [14], which was created using programming modules from STAN [10] and HSP-R. In the first phase, AltAlt uses the module from STAN to create a planning

graph, from which it extracts a heuristic function that is used to guide the backward hill-climbing search, which is performed in an HSP-R manner.

One of the latest planners in this category and the most effective according to the results of the AIPS-00 planning competition[1] is Hoffmann's FF planning system [7]. FF constructs a graph similar to this of GRAPHPLAN from which it extracts a sketch plan. The sketch plan is then used in a forward enforced hill-climbing search in two ways. Firstly, the length of the sketch plan is used as an estimate for the distance between the *Initial* state and the *Goals* and secondly a set of *helpful* actions is extracted which helps in cutting down the branching factor of the search.

Bi-directional search is a well-known search strategy mentioned in almost any textbook about Artificial Intelligence. However, it has not been broadly adopted as a search strategy. Especially in planning, there are only a few systems performing a combined search in both directions. The only bi-directional planners that have been developed, to the knowledge of the authors, are PRODIGY [19], NOLIMIT[18], FLECS[20] and RASPUTIN[6]. All of these planners have been developed by researchers of the Carnegie Mellon University's PRODIGY project and are based on the combination of goal-directed backward chaining with simulation of plan execution [18]. Although these planners perform some kind of search in both directions, they are actually forward-direction planners, which utilize the backward search as an action selection mechanism.

3. The Search Strategy of BP

The planners presented in the previous section have shown quite impressive performance and they have proved to be able to handle a large variety of difficult problems. However, they usually present variations in their efficiency among different domains or even between problems of the same domain.There are two main reasons that justify this behavior:

a) Although the heuristic functions constructed by all the planners are general, they seem to work better with specific domains.
b) There are domains and problems that clearly favor one of the two search directions (forward or backward).

The first argument, which is also a conclusion drawn from the experience of the authors, has been stated by Stone, Veloso and Blythe in [17]. The second argument is the main conclusion drawn by Bart Massey in an extensive study in the directions of planning presented in [11]. Bonet and Geffner have pushed the same argument one step further: "*Although we don't fully understand yet when HSP will run better than HSP-R, the results suggest nonetheless that in many domains a bi-directional planner combining HSP-R and HSP could probably do better than each planner separately*" [2]. The answer to the question posed by Bonet and Geffner above has been answered by Massey in [11], where the planning problems are discriminated into forward and backward problems, in the sense that strongly directed planners will find the problems of the opposite direction intractable.

Motivated by the conclusions stated above we developed BP, a heuristic state-space planner, which combines search in both directions. A part of the plan is

[1] A complete review of the participating systems, the domains and the results of the AIPS-00 competition can be found at the URL: http://www.cs.toronto.edu/aips2000/

constructed with the progression module (forward chainer) and the rest is constructed with the regression module (backward chainer). The sub-plan of the regression module is inversed and merged with that of the progression module in order to produce the final plan. However the case is not always that simple, because usually BP interleaves the execution of both modules several times before a solution is found. Details about the search strategy are presented later in this section but first we describe the progression and the regression search modules.

3.1 The Progression Module

The progression module employs a best-first search method starting from the initial state and moving forward trying to reach the goals with two main differences:

a) the size of the planning agenda is limited by an upper limit *SOF_AGENDA*. This means that if there are N states (N>SOF_AGENDA) only the SOF_AGENDA most promising (according to *h*) states will be stored and the rest will be pruned. This fact sacrifices completeness but it is necessary, since otherwise a lot of problems would become intractable.

b) The progression module will stop the search when it is not further possible to move to a state, the distance of which is not greater than the distance of the current state plus *T*. This part of the algorithm is crucial to the unified bi-directional search strategy, since the value of *T* determines how frequently will the algorithm change the search direction.

The progression module takes five arguments, which are: a) the initial state *I'* of the sub-problem, b) the goals *G'* of the sub-problem, c) the maximum size *SOF_AGENDA* of the planning agenda d) a threshold *T* declaring when should the search stop and e) a heuristic function *h* capable of estimating distances between states. The progression module returns a new state *S*, which is the state closer to *G'* that the module could find.

3.2 The Regression Module

The algorithm of the regression module is quite similar to the one of the progression module, since the search strategy is symmetric. The only differences are in the way of finding the applicable actions and forming the successor states. The regression module makes extensive use of binary mutual exclusions between facts. Two facts *p* and *q* are mutual exclusive, denoted as *mx(p,q)*, if no valid state contains both of them at the same time. Mutual exclusions are calculated in a way similar to the one they are calculated in GRAPHPLAN [1].

An action *A* is backward applicable to state *S* if *S* contains at least one add and no del effects of *A* and there are no mutual exclusions between facts of *S* and the facts added by *A*. State *S'* is produced from the backward application of action *A* to state *S* by removing the del effects of *A* from *S* and adding the add ones.

3.3 Combining the two Search Modules

The underlying framework of the bi-directional search strategy is based on a relatively simple idea. Usually, single-directional planners reach a point in the search process where the heuristic function becomes less informative. Two of the main reasons that justify this behavior are: a) the branching factor of the current sub-problem is too large for the heuristic to produce accurate estimates b) the sub-problem is much too complex and the heuristic function becomes obsolete as the search goes on, especially when it is constructed only once at the beginning.

In contrast, BP constructs the heuristic function in the backward direction and starts performing a forward directed search until it reaches a state S_B from where it is difficult to proceed. Then it reconstructs the heuristic function in the opposite (forward) direction and starts searching, in the opposite direction (backward), from the *Goals* towards S_B. If the backward search is also blocked after some steps in a state S_{B2}, BP will restart the planning process replacing the *Initial* state with S_B and the *Goals* with S_{B2}. The value of the *Threshold* is increased each time a search module returns without improving its initial state. The bi-directional search strategy of BP is outlined in Figure 1, where *St.plan* represents the plan from the *Initial* state to *St* for the progression module and the plan from the *Goals* to *St* for the regression one.

```
Search Algorithm of BP
Input I, G, Output Plan
Plan1=Plan2=[],S = I,F = G,Direction=Forward,Threshold=Init_Thr
While F⊄S
Begin
  If Direction = Forward
  begin
    Create backward heuristic function h_B
    St=Progression_module(S,F,MAX_SOF_AGENDA,Threshold,h_B)
    If St≠S Plan1=Plan1+St.plan, Threshold=Init_Thr, S=St
     Else Threshold=Threshold+STEP
     Direction = Backward
  end
  Else
  begin
    Create forward heuristic function h_F
    St=Regression_module(S,F,MAX_SOF_AGENDA,Threshold,h_F)
    If St≠F Plan2=Plan2+ St.plan,Threshold=Init_Thr,F=St
     Else Threshold=Threshold+STEP
     Direction = Forward
  End
end
Return Plan1+reversed(Plan2)
```

Figure 1: The Search algorithm of BP

There are two reasons that enable BP to face the problems stated above: a) the change in the direction enables BP to update the heuristic function, b) due to the adaptive way in which BP changes directions, it tends to solve the major part of the problem in the direction, which best fits it.

4. BP's Heuristic Functions

In order to test the efficiency of the bi-directional search strategy, we developed two, relatively simple, domain independent heuristics functions that were embedded in the BP planning system. The two heuristic functions are quite similar and are based on exactly the same idea, but the first one is used for the progression module and the other for the regression one. Note here, that both search modules of BP adopt a weighted A* search strategy, where the total cost of a state S is calculated as: $w_1*L(S)+w_2*h(S)$. In this formula, $L(S)$ is the number of steps needed to achieve state

S starting from the Initial state (Goals in case of regression), $h(S)$ is the value returned by the heuristic function and w_1 and w_2 are user-defined constants.

4.1 The Progression Heuristic Function

The heuristic function used for the progression module is similar to the one of the GRT planning system. The heuristic function is extracted from a leveled graph, similar to the one built by GRAPHPLAN. The graph consists of all the subgoals of the domain (the action levels of the GRAPHPLAN are omitted) that are generated from the Initial state, tagging them with a number K, identifying the minimum number of steps needed to generate them starting from the Goals.

The graph construction begins from the Goals of the problem (level 0) and proceeds backwards adding iteratively a new level *L* with all the subgoals that are generated by actions that are applicable at level *L*-1. An action *A* is applicable at level *L*-1, if at least one add-effect of *A* exists in *L*-1. For each action *A* that is applicable at level *L*-1, the algorithm computes a value *V* as the sum of the tags of all the facts in add(*A*). The facts in its precondition list are then added at level *L* and tagged with *V*+1 if they have not already been tagged with a smaller value than *V*+1. The expansion of the graph iterates until the graph reaches level L_{MAX}, where no more subgoals can be generated with a cost smaller than the one in its tags.

After the creation of the graph, which is done only once as long as the planner does not change direction, the tags of the facts are used to produce estimates for the distance between any state *S* in the domain and the goals, just by summing up the tags of the facts in *S*.

4.2 The Regression Heuristic Function

As stated earlier in this section, the regression heuristic function is similar to the progression one and just differs in the direction in which the graph is created. The graph for the regression is built starting from the facts in the Initial state (level 0) and proceeds forwards until it reaches a level L_{MAX}, where no more facts can be added to the graph with a cost smaller than the one in their tags. Here, an action *A* is applicable at level *L*, if all the preconditions of *A* exist in *L*.

4.3 Refining the heuristic functions with Goal Ordering

Goal ordering for planning has been an active research topic over the last years and a number of techniques have been successfully adopted by state-of-the-art planning systems. The research so far has been focused on two tasks: a) how to automatically extract as much information as possible about orderings among the goals of the problem, with minimum computational cost and b) how to use this information during planning. McCluskey and Porteous with their work on PRECEDE[12] proposed a method for identifying goal orderings between pairs of atomic facts, based on direct domain analysis. The more recent work of Koehler and Hoffman on GAM [8] have resulted in two techniques for identifying goal orderings, one based on domain analysis and another utilizing the information gained by the construction of a planning graph. The simplest and yet quite effective orderings extracted by these techniques have been described as *reasonable orders* and are based on the following idea:

> *"A pair of goals A and B can be ordered so that B is achieved before A if it isn't possible to reach a state in which A and B are both true, from a state in which A is true, without having to temporarily destroy A."* [15].

IPP [8] and FF[7] make use of reasonable orderings during planning through the construction of a goal agenda that divides the goals into an ordered set of sub-goals. The planners sequentially achieve the first sub-goal in the agenda, which has not yet been achieved. Experimental results have shown that the use of the goal agenda yields in significance improvement in terms of both planning time and plan quality.

BP adopts a slightly different method to compute reasonable orderings between goals, which is based on mutual exclusions between facts of the domain. Since the planner calculates the set of binary mutual exclusions, in order to use them for the regression phase, the overhead imposed by the calculation of reasonable orderings is negligible. Function *OB* (Ordered Before), which is outlined in Figure 2, is iteratively ran on every pair of goals in order to identify the possible orderings between the goals of the problem.

```
Function OB
Input: Goals a and b
Output: True (a should be ordered before b) or False
For each action O: a∈add(O)
begin
    MutexPre=false
    For each fact f: f∈prec(O)
        If mx(b,f)=true MutexPre=true
    If MutexPre = false return false
end
Return true
```

Figure 2: The OB Function

The orderings extracted by OB are used in the planning phase, in order to refine the results of the heuristic functions and not to divide the goals into sub-sets. More specifically, after the evaluation of a state *S* by one of the two heuristic functions, as exemplified by sub-sections 4.1 and 4.2, BP searches state *S* for possible violations of the goal orderings. Fact f of a state S is violating the goal ordering if:

$f \in$ Goals and $\exists$ goal g: g $\notin$ S and OB(g,f)=true

For every ordering violation in state S, the estimated distance between S and the Goals is increased by a constant number, since at a later point the ordering breaches will have to destroyed and re-achieved after the correct ordering has been reinstated.

5. Experimental Results

In order to test the efficiency of BP we implemented two additional planners: a) PMP (Progression Module Planner), a progression planner using the progression module and heuristic function of BP and b) RMP (Regression Module Planner), a regression planner using the regression module and heuristic function of BP. The search modules in PMP and RMP were slightly modified, so as to continue their search until a solution is found. The three planning systems were tested on a large variety of problems adopted by the recent AIPS-2000 planning competition.

The codes of the planners were based on the publicly available code of the second version of GRT and were implemented in C++. All the tests were run on a SUN ENTERPRISE 3000 parallel computer, with a SPARC-1 processor at 167 MHz and 256 MB of RAM. The underlying operating system was SUN Solaris 2.6 and the

programs were compiled by GNU C++ compiler. For the tests we have chosen the following configuration for the three planners:
`MAX_SOF_AGENDA`=200, `Init_Thr` = 2, `STEP` = 2, w_1=0.4 and w_2=1.0.

The three planners (PMP, RMP and BP) were tested on all problems of the *blocks world*, the *logistics*, the MIC-10 and the *freecell* domains used in the AIPS-00 planning competition. Tables 1, 2, 3 and 4 present the results of the tests. Columns 1,2 and 3 present the number of steps of the plan found by each planner and the time needed to solve the problem (in brackets). Note that short dashes mean that the problem could not be solved within the 180 seconds limit in CPU time set on all planners. Plan lengths written in bold note the minimum plan length found by the three planners. Note that due to space limitations, tables 1, 2, 3 and 4 present only a part of the tested problems.

5.1 Blocks world

It is clear from table 1 that the specific problems favor regression planners. RMP was able to solve 47% more problems than PMP producing in all problems shorter plans in much less time, while BP presented results quite similar to RMP. Specifically, BP solved 1 problem less than RMP, producing 16% longer plans, spending though 45% less time on average. BP clearly outrivaled PMP, producing 67% shorter plans and spending almost 20 times (1930%) less time on average.

5.2 Logistics

In the *logistics* problems the choice of planning direction didn't seem to play a very important role. RMP and BP solved more problems than PMP, but produced slightly worse plans than the latter. Specifically, BP produced 3% longer plans than PMP but solved 32% more problems in 7% less time on average. RPM solved the same number of problems with BP and was quite faster (9%) than the latter. However the plans it produced were 5% longer than those of BP.

Prob	PMP	RMP	BP
4-0	**6** (40)	**6** (40)	**6** (60)
5-0	**12** (790)	**12** (100)	**12** (110)
6-0	42 (14790)	**12** (140)	18 (270)
6-2	54 (15820)	24 (640)	**22** (260)
7-0	44 (12930)	**20** (410)	22 (390)
7-1	70 (13990)	**22** (430)	24 (440)
7-2	106 (42890)	**20** (530)	22 (410)
8-1	88 (27360)	**20** (490)	30 (850)
8-2	18 (250)	**16** (410)	**16** (390)
9-1	-	**30** (5150)	**30** (3570)
9-2	-	**26** (5130)	28 (1740)
10-1	-	**38** (21350)	42 (7490)
10-2	-	-	114 (40200)
11-0	62 (5270)	**34** (4730)	78 (8490)
11-1	-	**30** (2080)	-
11-2	-	-	**220** (125030)
12-0	-	**34** (5040)	48 (8680)
12-1	-	**38** (18380)	-
13-1	-	-	-
14-0	-	**38** (8170)	-
14-1	-	**36** (5910)	-
17-0	-	-	**50** (59280)

Table 1: Plan length and solution time (in msec) for Blocks world problems

Prob	PMP	RMP	BP
4-0	**20** (150)	23 (240)	**20** (240)
5-0	**27** (220)	32 (350)	**27** (350)
6-0	**25** (210)	33 (440)	28 (450)
7-1	62 (10730)	51 (1340)	**50** (1830)
8-0	**31** (640)	41 (1230)	37 (1340)
9-1	**32** (590)	34 (1230)	**32** (1320)
10-0	74 (29650)	54 (3790)	**50** (5010)
12-0	**45** (4590)	51 (3820)	**45** (4960)
13-1	-	83 (17620)	**81** (24160)
14-0	-	**78** (15490)	79 (31140)
14-1	**76** (5970)	93 (20410)	87 (30120)
15-0	112 (70170)	**95** (21480)	106 (38620)
15-1	**76** (20240)	85 (20070)	82 (24410)
16-0	-	**109** (35820)	112 (47130)
16-1	**83** (10330)	108 (40890)	85 (16430)
17-0	-	116 (41350)	**114** (53130)
17-1	-	**120** (45650)	**120** (85490)
18-1	104 (90900)	**101** (46410)	102 (59810)
19-1	-	119 (70160)	**113** (89260)
20-0	-	**127** (73580)	138 (116440)
20-1	**110** (22620)	123 (66510)	112 (33480)

Table 2: Plan length and solution time (in msec) for Logistics problems

5.3 MIC-10

MIC-10 is a domain that clearly favored progression planners, as shown by the experimental results. RMP was unable to solve problems harder than s4-1, while PMP and BP solved almost every problem of the domain. We tried to increase the size of the planning agenda for RMP and as a result the planner was able to solve a few more problems (up to s5-2) but this had a negative impact on planning time. Concerning the other two planners, BP clearly outperformed PMP by solving 7.5% more problems in 35% less time on average and by producing 10% shorter plans.

5.4 Freecell

Like the *logistics* domain, *freecell* does not clearly favor a specific planning direction. However PMP seemed to perform better than RMP and this is probably due to the fact that there is too much implied information that is omitted from the goals. In this domain BP solved 3 problems more than RMP and 2 less than PMP, producing plans of lower quality (approx. 6% longer plans) than both RMP and PMP. However, concerning planning time, BP clearly outperformed the other two, needing 35% less time than PMP and 614% less time than RMP on average.

Prob	PMP	RMP	BP
S1-0	**4** (0)	**4** (10)	**4** (20)
S3-0	12 (40)	**10** (320)	11 (40)
S4-0	16 (100)	-	**15** (100)
S10-0	40 (1980)	-	**36** (1390)
S11-0	41 (2400)	-	**39** (1580)
S12-0	48 (4280)	-	**41** (2500)
S15-0	60 (8710)	-	**52** (5130)
S17-0	67 (12840)	-	**62** (7880)
S17-4	65 (14280)	-	**57** (8040)
S18-4	70 (16050)	-	**62** (9140)
S19-0	-	-	**66** (11520)
S19-2	74 (26090)	-	**64** (11990)
S19-3	-	-	**67** (12100)
S19-4	77 (21910)	-	**66** (11570)
S20-0	-	-	**70** (16560)
S20-1	84 (28640)	-	**71** (13450)
S20-2	79 (23800)	-	**65** (12850)
S20-3	-	-	**72** (14770)
S20-4	-	-	**70** (15570)

Table 3: Plan length and solution time (in msec) for MIC-10 problems

Prob	PMP	RMP	BP
2-1	**9** (3920)	**9** (37660)	11 (4240)
2-2	**8** (3910)	**8** (34110)	9 (4150)
2-3	**8** (3510)	**8** (38860)	9 (3940)
2-4	**8** (4110)	**8** (32920)	9 (4150)
2-5	**9** (3930)	**9** (33770)	11 (4390)
3-1	18 (18990)	**15** (88360)	18 (43870)
3-2	**17** (20190)	19 (100650)	19 (15000)
3-3	16 (30130)	19 (90870)	**15** (10580)
3-4	15 (14950)	**13** (71910)	**13** (9990)
3-5	**16** (38760)	**16** (83530)	17 (19310)
4-1	**27** (106590)	-	28 (93750)
4-2	24 (25090)	**21** (150020)	-
4-3	**28** (78620)	-	38 (86140)
4-4	26 (67300)	**19** (127580)	-
4-5	30 (100620)	-	**24** (27880)
5-1	-	-	-
5-2	**28** (159790)	-	-
5-3	-	-	**46** (152930)
5-4	**29** (85080)	-	39 (153630)

Table 4: Plan length and solution time (in msec) for Freecell problems

6. Domain Analysis through Planning Graphs

The relaxed planning graphs built by the heuristic functions of BP, as described in sections 4.1 and 4.2, can be used as a means for extracting valuable information about the structure of the domain. This information can then be used in various ways, such as a) removing redundant information from the definition of the problem and thus cutting down the branching factor and b) identifying independent sub-goals that can be solved in parallel.

6.1 Simplifying the Definition of the Problem

The simplification of the problem's definition concerns facts of the *Initial* state that are useless for the planning process. For example, in the *logistics* domain the *Initial* state may contain facts noting the initial location of packages, which do not need to be

moved. This information is present in the description of the world, for means of completeness, but increases the branching factor of the problem with useless actions.

BP employs a simple but efficient method for eliminating useless facts from the *Initial* state, which is based on the backward graph built by the progression heuristic function. After the initial construction of the backward graph, the method eliminates all the facts of the *Initial* state that do not appear in the graph. If a fact *f* does not appear in the backward graph, there is no way to reach a state *S* where $f \in S$, by regressing the *Goals*. This means that the facts added by the actions that have *f* in their preconditions, are not present in the graph too and therefore *f* does not contribute at all in the process of reaching the *Goals* of the problem. So it is safe to remove it from the *Initial* state without jeopardizing completeness.

There is no actual overhead imposed by the above method, since the backward graph is built by the progression heuristic function no matter if the method for eliminating useless facts is applied or not. As far as the efficiency of the method is concerned, the method is not complete, in the sense that it does not identify all the forms of information that could be safely removed from the definition of the problem.

6.2 Identifying Independent Sub-Goals

Motivated by the results of the method for eliminating useless facts from the *Initial* state, we developed a second method for identifying independent sub-goals that can be solved in parallel. Given the definition of a problem $<I,A,G>$, the method iteratively builds *N* (size of *G*) backward planning graphs, removing at each time one of the goals in *G*. It follows from the method in the previous sub-section that if we remove goal G_k from *G*, then the facts in *I* (noted as I_k), that do not appear in the backward graph of $G-\{G_k,\}$ are closely related to G_k and they are not needed for the achievement of the rest of the goals. After the creation of the graphs, the algorithm comes up with a number *L* of sets of the form $<I_k,G_k>$, which indicate that the set I_K of initial facts is only needed for the achievement of goal G_K.

Consider, for example, a *logistics* problem with a city consisting of two locations (`airport` and `center`), two trucks (`tr1` and `tr2`) and three packages (P_1, P_2 and P_3) that have to be moved. The initial state and the goals of the problem are:

I={at(P_1,center),at(P_2,center),at(P_3,center),at(tr1,center),at(tr2,center)}

G={at(P_1,airport),at(P_2,airport),at(P_3,center)}

The independent goals and the sub-sets of G used for the graphs are the following:

G_1=at(P_1,airport), G-{G_1}={at(P_2,airport),at(P_3,center)}

G_2=at(P_2,airport), G-{G_2}={at(P_1,airport),at(P_3,center)}

G_3=at(P_3,center), G-{G_3}={at(P_1,airport),at(P_2,airport)}

The three subsets of the *Initial* state that are extracted from the backward graphs are the following:

I_1={at(P_1,center)}

I_2={at(P_2,center)}

I_3={at(P_3,center)}

So the sets extracted by our method are the following:

S_1=<{at(P_1,center)}, at(P_1,airport)>

S_2=<{at(P_2,center)}, at(P_2,airport)>

S_3=<{at(P_3,center)}, at(P_1,center)>

At a first step, these sets can be used for further elimination of redundant facts from the problem's definition. If for a given set $<I_k,G_k>$, the size of I_k is equal to 1 and

I_k $\{G_k\}$, it is safe to remove G_k from G and I_k from I and discard $<I_k,G_k>$. This is exactly the case with set S_3 of the previous example, where it is obvious that fact `at(P3,center)` could be safely removed from I and G.

The second use of the sets of the form $<I_k,G_k>$ is for dividing G in sub-goals the plans of which that can be combined in a parallel plan. The benefits of such a method are: a) a speedup in the planning process since tackling each sub-problem independently is usually easier than tackling the whole problem as one and b) a parallel plan which will probably be executed in less time than any sequential one.

The first argument is also supported by experimental results. We developed a planner, named BP-SP (Bi-directional Planner for Sub Problems), which uses the above method for dividing the initial problem into a number of sub-problems and then uses BP for solving the sub-problems sequentially. Supposing that we have L sets of the form $<I_k,G_k>$ and I and G are the initial state and the goals respectively, BP-SP starts with the sub-problem $<I',A,G'>$ where:

$$I' = I - \bigcup_{k=2}^{L} I_k \ , \ G' = \{G_1\}$$

If BP manages to solve $<I',A,G'>$ it will encounter a state S' where $G' \subseteq S'$. S' will be used for the next iteration of BP-SP to form the new sub-problem $<I'',A,G''>$ as follows:

$I''=S'-\{G_1\} \cup I_2$, $G''=\{G_2\}$

This process is repeated L times and the overall plan returned by BP-SP is constructed by concatenating the plans of the sub-problems.

The two planners (BP and BP-SP) were tested on all 80 *logistics* problems of the AIPS-00 planning competition. BP-SP managed to solve 66 problems, 20 more than BP, and it was almost 4 times faster on average. The resulted plans were approximately 30% longer but this could be probably overcome by a planner utilizing the fact that the sub-plans of BP-SP can be combined in a parallel plan. It lies in our future plans, to enhance BP with the ability to handle parallel problems.

7. Conclusions and Future Work

It is a generally accepted fact that there are certain domains or certain problems of domains that can be tackled more efficiently by forward planners and others that can be tackled more efficiently by backward ones. The matter of direction in planning is an active field of research and yet no clear answer has been given to the question of which direction should be generally preferred. This paper proposed BP, a hybrid planning system that combines search in both directions. BP changes the search direction in an adaptive way, which enables it to solve the major part of the problems in the direction that best fits them.

BP has been tested on a variety of problems used in the AIPS-00 competition and has been compared to two single-direction planners (a forward and a backward) that utilize the same heuristic function and optimization techniques. Experimental results have shown that BP has a stable performance on all domains, outperforming, in general, both of the single-direction planners.

It is in our future plans to develop a more sophisticated heuristic function and embody it in BP, along with several optimization techniques extracted from automatic domain analysis. Our experience with BP has shown that a large amount of useful

information can be extracted from the combination of planning graphs in both directions and this information can be used to construct efficient optimization techniques, such as the two methods discussed in section 6.

8. References

[1] Blum, L., and Furst M., 1995, Fast planning through planning graph analysis, In *Proc., 14th Int. Joint Conference on Artificial Intelligence*, Montreal, Canada, pp. 636-642.

[2] Bonet, B. and Geffner, H. 1999, Planning as Heuristic Search: New Results, In Proceedings, ECP-99, Durham UK.

[3] Bonet, B. and Geffner, H. 2000, Planning as Heuristic Search, Artificial Intelligence, forthcoming.

[4] Bonet, B., Loerincs, G., and Geffner, H., 1997, A robust and fast action selection mechanism for planning, In *Proc., 14th Int. Conference of the American Association of Artificial Intelligence (AAAI-97)*, Providence, Rhode Island, pp. 714-719.

[5] Fikes, R., and Nilsson, N., 1971, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, **2**: 189-208.

[6] Fink, E. and Blythe, J. 1998, A complete bidirectional planner, In proceedings, 4th International Conference on AI Planning Systems.

[7] Hoffmann, J. 2000, A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm, 12th Int. Symposium on Methodologies for intelligent Systems.

[8] Koehler, J. and Hoffmann, J. 2000, On Reasonable and Forced Goal Orderings and their Use in an Agenda-Driven Planning Algorithm, JAIR (12).

[9] Korf, R. 1998, Artificial intelligence search algorithms**,** CRC Handbook of Algorithms and Theory of Computation**,** Atallah, M. J. (Ed.), CRC Press, Boca Raton, FL, pp. 36-1 to 36-20

[10] Long, D. and Fox, M. 1998. Efficient Implementation of the Plan Graph in STAN, JAIR, 10, pp. 87-115.

[11] Massey, B. 1999, Directions In Planning: Understanding the Flow of Time in Planning, Available as a Technical Report from the University of Oregon.

[12] McCluskey, T. and Porteous, J. 1997, Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency, Artificial Intelligence, 95.

[13] McDermott, D. 1996, A Heuristic Estimator for Means-End Analysis in Planning, In Proceedings, AIPS-96

[14] Nguyen, X., Kambhampati, S. and Nigenda, R. 2000, AltAlt: Combining the advantages of Graphplan and Heuristics State Search, In Proceedings, 2000 International Conference on Knowledge-based Computer Systems, Bombay, India.

[15] Porteous, J. and Sebastia, L., 2000, Extracting and ordering Landmarks for Planning, In Proceedings, 18th Workshop of the UK Planning and Scheduling SIG

[16] Refanidis, I., and Vlahavas, I., 1999, *GRT:* A Domain Independent Heuristic for STRIPS Worlds based on Greedy Regression Tables, In *Proceedings, 5th European Conference on Planning*, Durham, UK, pp. 346-358.

[17] Stone, P., Veloso, M. and Blythe, J. 1994, The Need for Different Domain-Independent Heuristics, In proceedings, AIPS-94, Chicago, USA.

[18] Veloso, M. 1994, Planning and learning by Analogical Reasoning, Springer-Verlag.

[19] Veloso, M., Carbonell, J., Perez, A., Borrajo, D., Fink, E. and Blythe, J. 1995, Integrating Planning and Learning: The PRODIGY Architecture, Journal of Experimental and Theoretical Artificial Intelligence, 7(1).

[20] Veloso, M. and Stone, P. 1995, FLECS: Planning with a Flexible Commitment Strategy, JAIR (3).

Planning with Pattern Databases

Stefan Edelkamp

Institut für Informatik
Albert-Ludwigs-Universität
Georges-Köhler-Allee, Gebäude 51
D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

Abstract. Heuristic search planning effectively finds solutions for various benchmark planning problems, but since the estimates are either not admissible or too weak, optimal solutions are found in rare cases only. In contrast, heuristic pattern databases are known to significantly improve lower-bound estimates for optimally solving challenging single-agent problems like the 24-Puzzle and Rubik's Cube.
This paper studies the effect of pattern databases in the context of deterministic planning. Given a fixed state description based on instantiated predicates, we provide a general abstraction scheme to automatically create admissible domain-independent memory-based heuristics for planning problems, where abstractions are found in factorizing the planning space. We evaluate the impact of pattern database heuristics in A* and hill climbing algorithms for a collection of benchmark domains.

1 Introduction

General propositional planning is PSPACE complete [3], but when tackling specific benchmark planning instances, improving the solution quality usually reveals the intrinsic hardness of the problems. For example, plan existence of Logistic and Blocks World problem instances is polynomial, but minimizing the solution lengths for these planning problems is NP-hard [11]. On the other hand for some benchmark domains like *Sokoban* and *Mystery* even plan existence is NP-hard. Therefore, we propose a planner that is able to find optimal plans and, if challenging planning problems call for exponential resources, the planner approximates the optimal solution.

1.1 Optimal Planning Approaches

Graphplan [1] constructs a layered planning graph containing two types of nodes, action nodes and proposition nodes. In each layer the preconditions of all operators are matched, such that *Graphplan* considers instantiated actions at specific points in time. *Graphplan* generates partially ordered plans to exhibit concurrent actions and alternates between two phases: *graph extension* to increase the

search depth and *solution extraction* to terminate the planning process. *Graphplan* finds optimal parallel plans, but does not approximate solution lengths; it simply exhausts the given resources.

Another optimal planning approach is symbolic exploration simulating a breadth-first search according to the binary encoding of planning states. The operators unfold the initial state over time and an efficient theorem prover then searches for a satisfying truth assignment. A Boolean formula f_t describes the set of states reachable in t steps. If f_t contains a goal state, the problem is solvable with the minimal t as the optimal solution length.

Two approaches have been proposed. *Satplan* [17] encodes the planning problem with a standard representation of Boolean formulae as a conjunct of clauses. The alternative in the planner *Mips* [8] is to apply binary decision diagrams (BDDs); a data structure providing a unique representation for Boolean functions [2]. The BDD planning approach is in fact *reachability analysis* in model checking [4]. It applies to both deterministic and non-deterministic planning and the generated plans are optimal in the number of sequential execution steps. Usually, symbolic approaches cannot approximate except for recent preliminary results with domain abstractions [15] and with symbolic best-first search [6]. Though promising, the solution quality is not as good as in explicit search.

1.2 Heuristic Search Planning

Directed search is currently the most effective approach in classical AI-planning: four of five honored planning systems in the general planning track of the AIPS-2000 competition at least partially incorporate heuristic search. However, in traversing the huge state spaces of all combinations of grounded predicates, all planners rely on inadmissible estimates. The currently fastest deterministic planner, FF [13], solves a relaxed planning problem for each state to compute an inadmissible estimate. Furthermore, non-general pruning rules in FF such as *helpful action cuts* and *goal ordering cuts* help to avoid plateaus and local optima in the underlying hill-climbing algorithm. Completeness in undirected problem graphs is achieved by breadth-first searching improvements for the estimate and by omitting pruning in case of backtracks. Nevertheless, the daunting problem for FF are directed problem graphs with dead-ends from which its move committing hill-climbing algorithm cannot recover.

The best admissible estimate that has been applied to planning is the *max-pair* heuristic [10] implemented in the HSP planner. However, even by sacrificing optimality due to scaling, in AIPS-2000 this estimate was too weak to compete with the FF-heuristic. Moreover, own experiments with an improvement to *max-pair* according to a minimum matching on a graph weighted with fact-pair solution lengths were discouraging.

This paper proposes a pre-computed admissible heuristic that easily outperforms *max-pair* and by scaling the influence of the heuristic even the state-of-the-art FF-heuristic is beaten. To build the database we exhaustively search all state-to-goal distances in tractable abstractions of the planning state-space that serve as lower bound estimates for the overall problem. After studying the

pattern database framework, we present experiments with a sizable number of benchmark planning problems of AIPS-1998 and AIPS-2000 and draw concluding remarks.

2 Planning Space Representation

For the sake of simplicity we concentrate on the STRIPS formalism [9], in which each operator is defined by a precondition list P, an add list A, and a delete list D, but the presented approach can be extended to various problem description languages which can be parsed into a fixed state encoding. We refer to state descriptions and lists as sets/conjuncts of *grounded* predicates also called *facts* or *atoms*. This is not a limitation since all state-of-the-art planners perform grounding; either prior to the search or on the fly.

Definition 1. *Let F be the set of grounded predicates and O be a set of grounded STRIPS operators. The result S' of an operator $o = (P, A, D) \in O$ applied to a state $S \subseteq F$ is defined as $S' = (S \setminus D) \cup A$ in case $P \subseteq S$. Inverse STRIPS operators o^{-1} are given by $o^{-1} = ((P \setminus D) \cup A, D, A)$.*

We exemplify our considerations in the Blocks World domain of AIPS-2000, specified with the four operators `pick-up`, `put-down`, `stack`, and `unstack`. For example, the grounded operator `(pick-up a)` is defined as

$P = \{$`(clear a)`, `(ontable a)`, `(handempty)`$\}$,
$A = \{$`(holding a)`$\}$, and
$D = \{$`(ontable a)`, `(clear a)`, `(handempty)`$\}$

The goal of the instance 4-1 is defined by $\{$`(on d c),(on c a),(on a b)`$\}$ and the initial state is given by $\{$`(clear b) (ontable d)`, `(on b c)`, `(on c a)`, `(on a d)`$\}$. The first step to construct a pattern database is a domain analysis prior to the search. The output are *mutex groups* of mutually exclusive facts. In every state (reachable from the initial state), exactly one of the atoms in each group will be true. In general this construction is not unique such that we minimize the state description length over all possible partitionings as proposed for the MIPS planning system [7]. In the example problem we find the following nine mutex-groups.

- $G_1 = \{$`(on c a)`, `(on d a)`, `(on b a)`, `(clear a)`, `(holding a)`$\}$,
- $G_2 = \{$`(on a c)`, `(on d c)`, `(on b c)`, `(clear c)`, `(holding c)`$\}$,
- $G_3 = \{$`(on a d)`, `(on c d)`, `(on b d)`, `(clear d)`, `(holding d)`$\}$,
- $G_4 = \{$`(on a b)`, `(on c b)`, `(on d b)`, `(clear b)`, `(holding b)`$\}$,
- $G_5 = \{$`(ontable a)`, `true`$\}$,
- $G_6 = \{$`(ontable c)`, `true`$\}$,
- $G_7 = \{$`(ontable d)`, `true`$\}$,
- $G_8 = \{$`(ontable b)`, `true`$\}$, and
- $G_9 = \{$`(handempty)`, `true`$\}$,

where `true` refers to the situation, when none of the other atoms is present in a given state description.

Definition 2. *Let* $G = \{G_1, \ldots, G_k\}$ *with* $G_i \subseteq F \cup \{\mathtt{true}\}$ *for* $i \in \{1, \ldots, k\}$ *be the set of* mutex groups*, i.e.* $f_i \neq f_j$ *for* $f_i \in G_i \backslash \{\mathtt{true}\}$ *and* $f_j \in G_j \backslash \{\mathtt{true}\}$*. A* state *is a conjunct* $f_1 \wedge \ldots \wedge f_k$ *of facts* $f_i \in G_i$*,* $i \in \{1, \ldots, k\}$*. All represented states span the planning space*[1] $\mathcal{P}$*.*

3 Pattern Databases

A recent trend in single-agent search is to calculate the estimate with heuristic pattern databases (PDBs) [5]. The idea is to generate heuristics that are defined by distances in space abstractions. PDB heuristics are consistent[2] and have been effectively applied to solve challenging $(n^2 - 1)$-Puzzles [19] and Rubik's Cube [18]. In the $(n^2 - 1)$-Puzzle a pattern is a collection of tiles and in Rubik's Cube either a set of edge-*cubies* or a set of corner-*cubies* is selected.

For all of these problems the construction of the PDB has been implemented problem-dependently, i.e. by manual input of the abstraction for the puzzles and its storage by suitable perfect hash functions. In contrast, we apply the concept of PDBs to general problem-independent planning and construct pattern databases fully automatically.

3.1 State Abstractions

State space abstractions in the context of PDBs are concisely introduced in [12]: A state is a vector of fixed length and operators are conveniently expressed by label sets, e.g. an operator mapping $\langle A, B, \_ \rangle$ to $\langle B, A, \_ \rangle$ corresponds to a transposition of the first two elements for any state vector of length three. The state space is the transitive closure of the seed state S_0 and the operators O. A *domain abstraction* is defined as a mapping ϕ from one label set L to another label set K with $|K| < |L|$ such that states and operators can be simplified by reducing the underlying label set. A *state space abstraction* of the search problem $\langle S_0, O, L \rangle$ is denoted as $\langle \phi(S_0), \phi(O), K \rangle$. In particular, the abstraction mapping is non-injective such that the abstract space (which is the image of the original state space) is therefore much smaller than the original space.

The framework in [12] only applies to certain kinds of permutation groups, where in our case the abstract space is obtained in a more general way, since abstraction is achieved by projecting the state representation.

[1] The planning space $\mathcal{P}$ is in fact smaller than the set of subsets of grounded predicates, but includes the set of states reachable from the initial state.

[2] Consistent heuristic estimates fulfill $h(v) - h(u) + w(u, v) \geq 0$ for each edge (u, v) in the underlying, possibly weighted, problem graph. They yield monotone merits $f(u) = g(u) + h(u)$ on generating paths with weight $g(u)$. Admissible heuristics are lower bound estimates which underestimate the goal distance for each state. Consistent estimates are indeed admissible.

Definition 3. *Let F be the set of grounded predicates. A* planning space abstraction *ϕ is a mapping from F to $F \cup \{\texttt{true}\}$ such that for each group G either for all $f \in G : \phi(f) = f$ or for all $f \in G : \phi(f) = \texttt{true}$.*

Since planning states are interpreted as conjuncts of facts, ϕ can be extended to map each planning state of the original space $\mathcal{P}$ to one in the abstract space $\mathcal{A}$. In the example problem instance we apply two planning space abstractions ϕ_{odd} and ϕ_{even}. The mapping ϕ_{odd} assigns all atoms in groups of odd index to the trivial value `true` and, analogously, ϕ_{even} maps all fluents in groups with even index value to `true`. All groups not containing a atoms in the goal state are also mapped to `true`[3]. In the example, the goal is partitioned into $\phi_{even}(G) = \{\texttt{(on c a)}\}$ and $\phi_{odd}(G) = \{\texttt{(on a b)}, \texttt{(on d c)}\}$, since the groups G_4 to G_9 are not present in the goal description.

Abstract operators are defined by intersecting their precondition, add and delete lists with the set of non-reduced facts in the abstraction. This accelerates the construction of the pattern table, since several operators simplify.

Definition 4. *Let ϕ be a planning space abstraction and $\delta_\phi(S_1, S_2)$ be the graph-theoretical shortest path between to two states S_1 and S_2 in $\mathcal{A}$. Furthermore, let S_0 be the start and S_t be the set of goal states in $\mathcal{P}$. A* planning pattern database (PDB) *according to ϕ is a set of pairs, with the first component being an abstract planning state S and the second component being the minimal solution length in the abstract problem space, i.e.,*

$$\text{PDB}(\phi) = \{(S, \delta_\phi(S, \phi(S_t))) \mid S \in \mathcal{A}\}.$$

PDB(ϕ) is calculated in a breadth-first traversal starting from the set of goals in applying the inverse of the operators. Two facts about PDBs are important. When reducing the state description length n to αn with $0 < \alpha < 1$ the state space and the search tree shrinks exponentially; e.g. 2^n bit vectors correspond to an abstract space of $2^{\alpha n}$ elements.

The second observation is that once the pattern database is calculated, accessing the heuristic estimate is fast by a simple table lookup (cf. Section 3.3). Moreover, PDBs can be re-used for the case of different initial states. *PDB*(ϕ_{even}) and *PDB*(ϕ_{odd}) according to the abstractions ϕ_{even} and ϕ_{odd} of our example problem are depicted in Table 1. Note that there are only three atoms present in the goal state such that one of the pattern databases only contains patterns of length one. Abstraction ϕ_{even} corresponds to G_1 and ϕ_{odd} corresponds to the union of G_2 and G_4.

3.2 Disjoint Pattern Databases

Disjoint pattern databases add estimates according to different abstractions such that the accumulated estimates still provide a lower bound heuristic.

[3] To include mutex-groups into PDB calculations which are not present in the goal state, we may generate *all possible instances* for the fact set. In fact, this is the approach that is applied in our implementation.

((clear a),1)	((on d c)(clear b),1)	((on a b)(clear c),1)
((holding a),2)	((on d c)(holding b),2)	((clear c)(clear b),2)
((on b a),2)	((on d c)(on d b),2)	((on a b)(holding c),2)
((on d a),2)	((on a c)(on a b) ,2)	((clear c)(holding b),3)
	((clear b)(holding c),3)	((on a c)(clear b),3)
	((on d b)(clear c),3)	((holding c)(holding b),4)
	((on b c)(clear b),4)	((on a c)(holding b),4)
	((on c b)(clear c),4)	((on d b)(holding c),4)
	((on a c)(on d b),4)	((on b c)(holding b),5)
	((on a b)(on b c),5)	((on d b)(on b c),5)
	((on c b)(holding c),5)	((on a c)(on c b),5)
	((on c b)(on d c),5)	

Table 1. Pattern databases $PDB(\phi_{even})$ and $PDB(\phi_{odd})$ for the example problem.

Definition 5. *Two pattern databases* PDB(ϕ_1) *and* PDB(ϕ_2) *are* disjoint*, if the sum of respective heuristic estimates always underestimates the overall solution length, i.e.,* $\delta_{\phi_1}(\phi_1(S), \phi_1(S_t)) + \delta_{\phi_2}(\phi_2(S), \phi_2(S_t)) \leq \delta(S, S_t), \forall S \in \mathcal{P}$.

PDBs are not always disjoint. Suppose that a goal contains two atoms p_1 and p_2, which are in groups 1 and 2, respectively, and that an operator o makes both p_1 and p_2 true. Then, the distance under abstraction ϕ_1 is 1 (because the abstraction of o will make p_2 in group 2 true in one step) and the distance under ϕ_2 is also 1 (for the same reason). But the distance in the original search space is also 1.

Definition 6. *An* independent abstraction set *I is a set of group indices such that no operator affects both atoms in groups in I and atoms in groups that are not in I. The according abstraction ϕ_I that maps all atom groups not in I to true is called an* independent abstraction.

Theorem 1. *A partition of the groups into independent abstractions sets yields disjoint pattern databases.*

Proof. Each operator changes information only within groups of a given partition and an operator of the abstract planning space contributes one to the overall estimate only if it changes facts in available fact groups. Therefore, by adding the solution lengths of different abstract spaces each operator on each path is counted at most once.

For some domains like Logistics operators act locally according to any partition into groups so that the precondition of Theorem 1 is trivially fulfilled.

3.3 Perfect Hashing

PDBs are implemented as a (perfect) hash table with a table lookup in time linear to the abstract state description length.

According to the partition into groups a perfect hashing function is defined as follows. Let $G_{i_1}, G_{i_2}, \ldots, G_{i_k}$ be the selected groups in the current abstraction and $\mathit{offset}(k)$ be defined as $\mathit{offset}(k) = \prod_{l=1}^{k} |G_{i_{l-1}}|$ with $|G_{i_0}| = 1$. Furthermore, let $\mathit{group}(f)$ and $\mathit{position}(f)$ be the group index and the position in the group of fact f, respectively. Then the perfect hash value $\mathit{hash}(S)$ of state S is

$$\mathit{hash}(S) = \sum_{f \in S} \mathit{position}(f) \cdot \mathit{offset}(\mathit{group}(f)).$$

Since perfect hashing uniquely determines an address for the state S, S can be reconstructed given $\mathit{hash}(S)$ by extracting all corresponding group and position information that define the facts in S. Therefore, we establish a good compression ratio, since each state in the queue for the breadth-first search traversal only consumes one integer. The breadth-first-search queue is only needed for construction and the resulting PDB is a plain integer array of size $\mathit{offset}(k+1)$ encoding the distance values for the corresponding states, initialized with ∞ for patterns that are not encountered. Some states are not generated, since they are not reachable, but the above scheme is more time and space efficient than ordinary hashing storing the uncompressed state representation. Since small integer elements consume only a few bytes, on current machines we may generate PDBs of a hundred million entries and more.

3.4 Clustering

In the simple example planning problem the combined sizes of groups and the total size of the generated pattern databases $PDB(\phi_{even})$ and $PDB(\phi_{odd})$ differ considerably. Since we perform a complete exploration in the generation process, in larger examples the requirements in time and space resources for computing PDBs might be exhausted. Therefore, an automatic way to find a suitable balanced partition according to given memory limitations is required. Instead of a bound on the total size of all PDBs together, we globally limit the size of each pattern database, which is in fact the number of expected states. The restriction is not crucial, since the number of different pattern databases is small in practice. The threshold is the parameter to tune the quality of the estimate. Obviously, large threshold values yield optimal estimates in small problem spaces.

We are confronted with a Bin-Packing variant: Given the sizes of groups, the task is to find the minimal number of pattern databases such that the sizes do not exceed a certain threshold value. Notice that the group sizes are multiplied in order to estimate the search space size. However, the corresponding encoding lengths add up. Bin-Packing is NP-hard in general, but good approximation algorithms exist. In our experiments we applied the best-fit strategy.

4 Results

All experimental results were produced on a Linux PC, Pentium III CPU with 800 MHz and 512 MByte. We chose the most efficient domain-independent planners as competitors. In Logistics, the program FF is chosen for comparison and

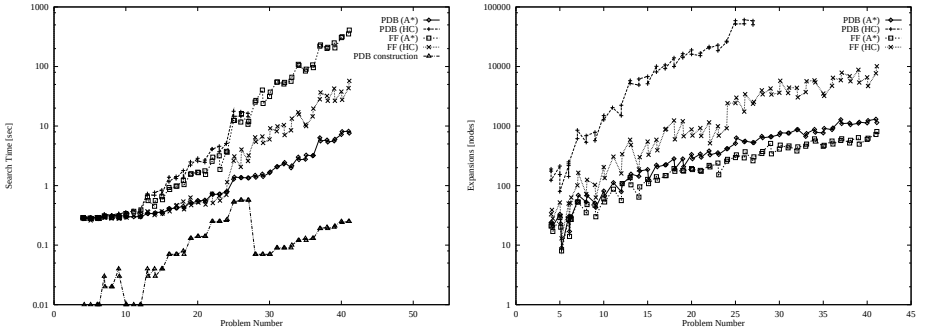


Fig. 1. Time performances and numbers of expansions of A* and hill climbing in the Logistics domain with respect to the PDB and FF heuristic on a logarithmic scale. PDB construction time is included in the overall search time.

in Blocks World, the pattern database approach is compared to the optimal planner *Mips*.

4.1 Logistics

We applied PDBs to Logistics and solved the entire problem set of AIPS-2000. The largest problem instance includes 14 trucks located in one of three locations of the 14 cities. Together with four airplanes the resulting state space has a size of about $3^{14} \cdot 14^4 \cdot 60^{42} \approx 8.84223 \cdot 10^{85}$ states. All competing planners that have solved the entire benchmark problem suite are (enforced) hill-climbers with a variant of the FF heuristic and the achieved results have about the same characteristics [14]. Therefore, in Table 1 we compare the PDB approach with the FF-heuristic. In the enforced hill climbing algorithm we allow both planners to apply branching cuts, while in A* we scale the influence of the heuristic with a factor of two. The effects of scaling are well-known [22]: weightening A* possibly results in non-optimal solution, but the search tends to succeed much faster. In the AIPS-2000 competition, the scaling factor 2 has enhanced the influence of the *max-pair* heuristic in the planner HSP. However, even with this improvement it solves only a few problems of this benchmark suite.

The characteristics of the PDB and FF heuristics in Figure 1 are quite different. The number of expanded nodes is usually larger for the former one but the run time is much shorter. A* search with PDBs outperforms FF with hill climbing *and* branching cuts. The savings are about two orders of magnitude with respect to FF and A* and one order of magnitude with respect to FF and hill climbing, while the effect for the number of expansions is the exact opposite. In the example set the average time for a node expansion in PDB-based planning is smaller by about two orders of magnitude compared to FF.

On the other hand, in larger problem instances enforced hill climbing according to the PDB heuristic generates too many nodes to be kept in main memory. In a few seconds the entire memory resources were exhausted. This

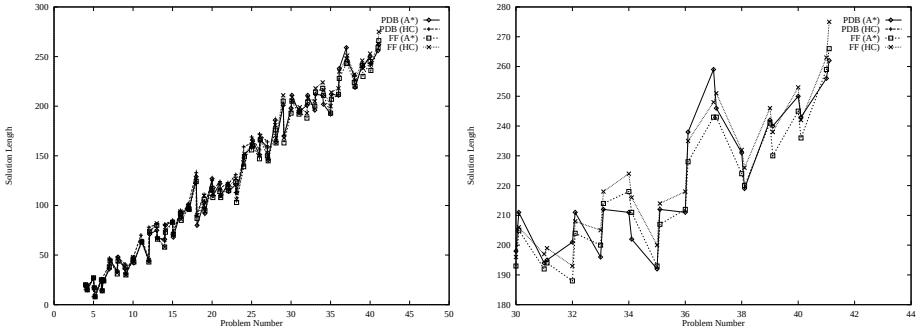


Fig. 2. Overall and magnified solution quality of A* and enforced hill climbing in the Logistics domain with respect to to the PDB and FF heuristic.

suggests applying memory limited search algorithm like thresholding in IDA* and alternative hashing strategies to detect move transpositions in high search depths.

We summarize that hill climbing is better suited to the FF heuristic while weighted A* seems to perform better with PDBs. The solution qualities are about the same as Figure 2 deptics. Even magnification to larger problem instances fails to establish a clear-cut winner.

4.2 Blocks World

Achieving approximate solutions in Blocks World is easy; 2-approximations run in linear time [24]. Moreover, different domain-dependent cuts drastically reduce the search space. Hence, TALPlanner [20] with hand-coded cuts and FF with hill climbing, helpful action and goal ordering cuts find good approximate solutions to problems with fifty Blocks and more. FF using enforced hill climbing without cuts is misguided by its heuristic, backtracks and tends to get lost in local optima far away from the goal. We concentrate on optimal solutions for this domain. Since any n-Tower configuration is reachable from the initial state state, the state space grows exponentially in n, and indeed, optimizing Blocks World is NP-hard. *Graphplan* is bounded to about 9 blocks and no optimal heuristic search engine achieves a better performance, e.g. HSP with *max-pair* is bounded to about 6-7 blocks. Model checking engines like BDD exploration in *Mips* and iterative Boolean satisfiability checks in *Satplan* are best in this domain and optimally solve problems with up to 12-13 blocks. Table 3 depicts that PDBs are competitive and that the solution lengths match.

Moreover, better scaling in time seems to favor PDB exploration. However, in both approaches space consumption is more crucial than time. In the bidirectional symbolic breadth-first search engine of Mips the BDD sizes grow very rapidly and large pattern databases with millions of entries still lead to millions of node expansions. When searching for optimal solutions to 13-block benchmark

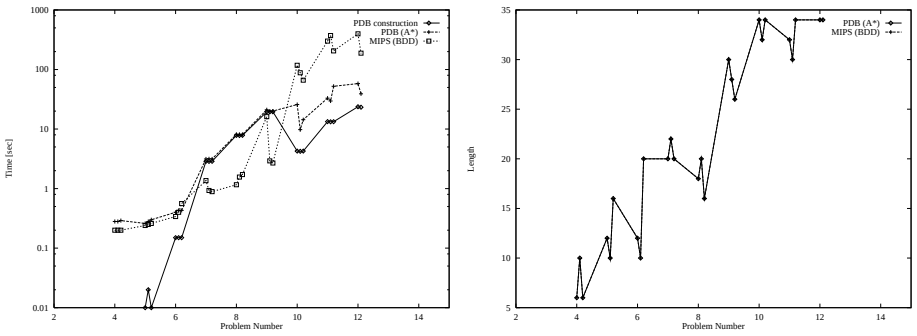


Fig. 3. Time performance and solution quality of BDD expoloration and optimal PDB planning in Blocks World. PDB construction time is included in the overall search time.

problems this thrashes the memory resources in both planning approaches. In summary, optimal solving Blocks World is still hard for general planning engines.

4.3 Other Domains

Gripper (AIPS-1998) spans an exponentially large but well-structured search space such that greedy search engines finds optimal solutions. On the other hand, Gripper is known to be hard for *Graphplan*. Both FF with hill-climbing and cuts and PDB with weighted A* find optimal solutions in less than a second.

Like Logistics, the NP-hard [11] *Mystery* domain (AIPS-1998) is a transportation domain on a road map. Trucks are moving around this map and packages are being carried by the mobiles. Additionally, various *capacity* and *fuel constraints* have to be satisfied. Mystery is particularly difficult for heuristic search planning, since some of the instances contain a very high portion of undetected dead-ends [14]. In contrast to the most effective heuristic search planner GRT [23], the PDB planning algorithm does not yet incorporate manual reformulation based on explicit representation of resources. However, experiments show that PDB search is competitive: problems 1-3, 9, 11, 17, 19, 25-30 were optimally solved in less then 10 seconds, while problem 15 and 20 required about 5 and 2 minutes, respectively. At least problem 4,7, and 12 are not solvable. Time performance and the solution qualities are better than in [23] Scaling reduces the number of node expansion in some cases but has not solved any new problem.

The start position of *Sokoban* consists of a selection of balls within a maze and a designated goal area into which the balls have to be moved. A man, controlled by the puzzle solver, can traverse the board and push balls onto adjacent empty squares. Sokoban has been proven to be PSPACE complete and spans a directed search space with exponentially many dead-ends, in which some balls cannot be placed onto any goal field [16]. Therefore, hill climbing will eventually encounter a dead-end and fail. Only overall search schemes like A*, IDA* or best-first prevent the algorithm from getting trapped. In our experiments we optimally solved all 52 automatically generated problems [21] in less than five seconds each. The

screens were compiled to PDDL with a one-to-one ball-to-goal mapping so that some problems become unsolvable. Since A* is complete we correctly establish unsolvability of 15 problems in the test set. Note that the instances span state spaces much smaller than the 90 problem suite considered in [16] with problems currently too difficult to be solved with domain independent planning.

As expected, additional results in Sokoban highlight that in contrast to the PDB-heuristic, the FF-heuristic, once embedded in A*, yields good but not optimal solutions. BDD exploration in Mips does find optimal solutions, but for some instances it requires over a hundred seconds for completion.

5 Conclusion

Heuristic search is currently the most promising approach to tackle huge problem spaces but usually does not yield optimal solutions. The aim of this paper is to apply recent progress of heuristic search in finding optimal solutions to planning problems by devising an automatic abstraction scheme to construct pre-compiled pattern databases.

Our experiments show that pattern database heuristics are very good admissible estimators. Once calculated our new estimate will be available in constant time since the estimate of a state is simply retrieved in a perfect hash table by projecting the state encoding. We will investigate different pruning techniques to reduce the large branching factors in planning. There are some known general pruning techniques such as *FSM pruning* [25], *Relevance Cuts* and *Pattern Searches* [16] that should be addressed in the near future.

Although PDB heuristics are admissible and calculated beforehand, their quality can compete with the inadmissible FF-heuristic that solves a relaxed planning problem for *every* expanded state. The estimates are available in a simple table lookup, and, in contrast to the FF-heuristic, A* finds optimal solutions. Weighting the estimate helps to cope with difficult instances for approximate solutions. Moreover, PDB heuristics in A* can handle directed problem spaces and prove unsolvability results.

One further important advantage of PDB heuristics is the possibility of a symbolic implementation. Due to the representational expressiveness of BDDs, a breadth-first search (BFS) construction can be completed with respect to larger parts of the planning space for a better quality of the estimate. The exploration yields a relation $H(\mathit{estimate}, \mathit{state})$ represented with a set of Boolean variables encoding the BFS-level and a set of variables encoding the state. Algorithm BDDA*, a symbolic version of A*, integrates the symbolic representation of the estimate [6]. Since PDBs lead to consistent heuristics the number of iterations in the BDDA* algorithms is bounded by the square of the solution length. Moreover, symbolic PDBs can also be applied to explicit search. The heuristic estimate for a state can be determined in time linear to the encoding length.

Acknowledgments We thank J. Hoffmann for the Sokoban problem generator, M. Helmert for eliminating typos, the anonymous referees for helpful comments, and P. Haslum for fruitful discussions on this research topic.

References

1. A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *IJCAI*, pages 1636–1642, 1995.
2. R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *DAC*, pages 688–694, 1985.
3. T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, pages 165–204, 1994.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
5. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.
6. S. Edelkamp. Directed symbolic exploration and its application to AI-planning. In *AAAI Symposium (Model-based Validation of Intelligence)*, pages 84–92, 2001.
7. S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *ECP*, LNCS, pages 135–147. Springer, 1999.
8. S. Edelkamp and M. Helmert. The model checking integrated planning system MIPS. *AI-Magazine*, 2001. To Appear.
9. R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, (2):189–208, 1971.
10. P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 140–149, 2000.
11. M. Helmert. On the complexity of planning in transportation and manipulation domains. Master's thesis, Computer Science Department Freiburg, 2001. Available from http://www.informatik.uni-freiburg.de/~ki/theses.html.
12. I. T. Hernádvögyi and R. C. Holte. Experiments with automatic created memory-based heuristics. In *SARA*, 2000.
13. J. Hoffmann. A heuristic for domain independent planning and its use in an enforced hill climbing algorithm. In *ISMIS*, LNCS, pages 216–227. Springer, 2000.
14. J. Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *IJCAI*, 2001. To appear.
15. R. Jensen and M. M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *JAIR*, 13, 2000.
16. A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, 1999.
17. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *AAAI*, pages 1194–1201, 1996.
18. R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI*, pages 700–705, 1997.
19. R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 2001. To appear (http://www.elsevier.nl/locate/artint).
20. J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and resources. In *ECAI*, pages 501–505, 2000.
21. Y. Murase, H. Matsubara, and Y. Hiraga. Automatic making of Sokoban problems. In *Pacific Rim Conference on AI*, 1996.
22. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
23. I. Refanidis and I. Vlahavas. Heuristic planning with resources. In *ECAI*, pages 521–525, 2000.
24. J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, pages 119–153, 2001.
25. L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI*, pages 756–761, 1993.

A Forward Search Planning Algorithm with a Goal Ordering Heuristic

Igor Razgon and Ronen I. Brafman

Computer Science Department
Ben-Gurion University, 84105, Israel
{irazgon,brafman}@cs.bgu.ac.il

Abstract. Forward chaining is a popular strategy for solving classical planning problems and a number of recent successful planners exploit it. To succeed, a forward chaining algorithm must carefully select its next action. In this paper, we introduce a forward chaining algorithm that selects its next action using heuristics that combine backward regression and goal ordering techniques. Backward regression helps the algorithm focus on actions that are relevant to the achievement of the goal. Goal ordering techniques strengthens this filtering property, forcing the forward search process to consider actions that are relevant at the *current* stage of the search process. One of the key features of our planner is its dynamic application of goal ordering techniques: we apply them on the main goal as well as on all the derived sub-goals. We compare the performance of our planner with FF – the winner of the AIPS'00 planning competition – on a number of well-known and novel domains. We show that our planner is competitive with FF, outperforming it on more complex domains in which sub-goals are typically non-trivial.

List of keywords: forward chaining, backward regression, goal ordering, relaxed problem

1 Introduction

Forward chaining is a popular strategy for solving classical planning problems. Early planning systems, such as GPS [12], used forward chaining but were quickly overcome by regression-based methods such as partial-order planning [17] and, more recently, GRAPHPLAN [2]. These methods were viewed as more informed, or focused. However, with the aid of appropriate heuristic functions, recent forward-chaining algorithms [3, 6] have been able to outperform other planners on many domains.

To succeed, a forward-chaining planner must be informed in its selection of actions. Ideally, the action chosen at the current state must bring us closer to the goal state. To achieve this goal, recent forward-chaining planners use new, improved heuristic functions in which regression techniques play an important role. The idea of backward regression through forward chaining was first explicitly stated by McDermott [13]. It was implicitly used in GPS and FF [6] and in its

more general form in GRAPHPLAN and its descendants [2, 9, 11] (as a polynomial structure constructed in a direction opposite to the main search direction). It was used also in [1] for relevance computation.

In this paper we extend regression-based relevance filtering techniques with a dynamic goal-ordering heuristics. This results in a planning algorithm – RO (Regression + Goal-Ordering) – that has a better chance of choosing actions that are relevant and timely. The backward regression heuristics used in our planner is somewhat similar to the one used in GPS. That is, we construct a sequence of subgoals, where the first subgoal is the main goal and the last subgoal is satisfied in the current state. The difference between RO and previous algorithms is in the way this subgoal sequence is generated. More specifically, given the current subgoal in the constructed sequence, the next subgoal is computed as follows: we select the proposition that we believe should be achieved *first* in the current subgoal – we use a goal ordering heuristics to make this selection. Then, we select an action that has this proposition as an add-effect. Finally, we add the preconditions of this chosen action to the beginning of the constructed sequence. Thus, we have a combination of a backward regression method with a goal ordering technique, where the ordering is computed *dynamically* for all subgoals of the sequence of goals generated by the backward regression process.

The combination of backward regression with goal ordering is based on the following intuition: One of the main goals of backward search in the context of forward chaining algorithm is to avoid considering irrelevant actions. [1]. However, when a relevant action is inserted in an inappropriate place in the plan, we either obtain a plan that is longer than necessary or we must perform expensive backtracking. By ordering subgoals, we strengthen the filtering property of backward regression and reduce the need for backtracking because we force the forward chaining process to consider actions that are relevant at the *current* stage of the search process.

Typically, goal ordering is done for the original goal only, and prior to the initiation of search – it is *static* in nature. Next, the problem is divided into smaller subproblems, each of which can be solved separately [12, 10]. Dynamic ordering of propositions is often seen in CSP problems. It was used in the context of GRAPHPLAN's backward search, viewed as a CSP problem, in [8]. This paper is among the first to use dynamic goal ordering in the planning process, and the particular dynamic goal ordering approach we introduce here is the main novel contribution of this paper.

The rest of this paper is organized as follows: Section 2 provides a short review of related work. Section 3 describes the proposed algorithm. In Section 4 we provide an empirical comparison between our planner and the winner of the AIPS 2000 planning competition, FF [6]. The main conclusion of our experimental analysis is that RO is competitive with FF, outperforming it on domains in which the subproblems obtained after ordering the top level goal are non-trivial. Finally, Section 5 concludes this paper.

2 Related Work

In this section we provide a short review of forward search algorithms and goal ordering methods and their use in planning. This will help provide some of the necessary background and will place this work in its proper context.

2.1 Recent forward search algorithms

HSP The HSP algorithm [3] is a forward chaining algorithm without backtrack. Each step, the action leading to the state with minimal approximated distance to the goal is chosen. This distance is approximated by solving a *relaxed* problem in which the delete effects of all actions were removed. Given this relaxed problem, we approximate the distance from the current state *cur* to a state s using the sum (or maximum) of the weights of the propositions that hold in s. The weight of a proposition p is 0 if it belongs to *cur*. Otherwise, $weight(p) = min_{all\ acts\ achieving\ p}\ 1 + dist(precond(act))$, i.e., one more than the minimal distance of the set of preconditions of actions that achieve p.

FF FF (Fast Forward) [6] is a forward search algorithm that selects an action at each step using an approximated distance measure. The distance from the current state to the goal is approximated by the *plan length* for achieving the goal in a relaxed problem induced by the current problem. FF uses GRAPHPLAN to solve the relaxed problem. Because the relaxed problem does not have del-effects, there are no mutually exclusive pairs of propositions and actions in the problem. Therefore, it can be solved in polynomial time. To make the measure near-optimal, various heuristics for decreasing the length of the extracted solution are applied.

The algorithm applies breadth-first search from the current state s until it finds a state s' that has a strictly better value. The ordering technique from [10], described below, is applied to make FF more effective. FF won the AIPS 2000 planning competition.

2.2 Goal ordering methods

There has been much work on goal ordering methods and their use in planning. Most of this work addresses the following four questions

1. How to order two propositions? [7, 10, 8]
2. How to derive a total order on propositions given an ordering on all pairs of propositions? [10]
3. How to use a total order on goal propositions in planning? [12, 10, 8]
4. How to incorporate propositions that are not in the top-level goal into the goal order? [14]

For our purpose, the first and the third questions are the most relevant.

An interesting classification of methods determining the order between two given propositions is introduced in [7]. According to it, we may conclude that $p < q$ if at least one of the following conditions holds.

1. **Goal subsumption.** To achieve q we must achieve p
2. **Goal clobbering.** When achieving p we destroy q if it existed in some previous state. An example of goal clobbering is the pair of propositions $on(1,2)$ and $on(2,3)$ in the BlocksWorld problem. $on(2,3)$ has to be achieved before $on(1,2)$, because $on(1,2)$ is destroyed in the process of achieving $on(2,3)$
3. **Precondition violation.** Achievement of q makes achievement of p impossible (dead-end).

Two criteria for ordering propositions based on the principle of goal clobbering are given in [10]. A modified version of the first of them is used is the proposed algorithm and described in detail in the next section.

A widespread method for using a goal ordering in planning is to iteratively apply the planning algorithm to achieve increasing subsets of the goal. For example if we have a goal ordering$(p_1, ..., p_n)$, then first we try to achieve a state s_1 in which $\{p_1\}$ holds, starting at the initial state. Next, we try to achieve $\{p_1, p_2\}$ starting at s_1, etc. The last plan achieves the required goal from $s_1, \ldots, s_{n-1}$. By concatenating the resulting plans, we get a complete plan.

3 Algorithm description.

We now proceed to describe the RO planning algorithm. In Section 3.1 we describe the algorithm and its action selection heuristic. In Section 3.2 we provide more details on some of the subroutines used during action selection. In Section 3.3, we discuss some optimization implemented in the current version of RO. Finally, in Section 3.4., we demonstrate the work of RO on a running example.

3.1 The Proposed Algorithm and its Action Selection Heuristic

RO is a forward chaining planner with chronological backtracking. It receives a domain description as input and an integer n denoting search-depth limit (the default value of n is 2000).

The first step of RO is to compute some data that will be useful during the search process. In particular RO constructs an approximated set of pairs of mutually exclusive propositions. It is known that exact computation of all mutual exclusive pairs of preconditions is not less hard than the planning problem [2]. Therefore, only approximation algorithms are acceptable in this case. RO obtains an approximate set of mutual exclusive pairs as the complement of a set of reachable pairs which is found by a modification of Reachable-2 algorithm [4].

Next, a standard depth-first search with chronological backtracking is performed (Figure 1). (Note, that to obtain the desired plan, we have to run this

function on the initial state and the empty plan). The heart of this algorithm is the heuristic selection of action that will be appended to the current plan (line 5). This heuristics is based on backward search without backtracks (backward regression). The depth of this regression phase, $MaxDeep$,is a parameter of the planner (the default value of $MaxDeep$ is 50). The code of this procedure is given in Figure 2.

As we can see from the code in Figure 2, this procedure builds a sequence of subgoals, starting with the main (i.e., original) goal as its first element. At each iteration, the current (last) subgoal in this sequence is processed as follows: its propositions are ordered (line 2) and the minimal proposition that is not satisfied in the current state (the **required** proposition) is selected (line 3). Next, an action achieving this proposition is chosen (line 4). If this action is feasible in the current state, it is returned. Otherwise, the set of preconditions of this action is appended to the subgoal sequence becoming the new current subgoal, and the process is repeated.

We see that RO combines goal ordering and backward regression techniques: each time a new subgoal is selected, its propositions are ordered, and this ordering is used to select the next action for the regression process. This combination is the main novel contribution of this work.

$ComputePlan(CurState, n, CurPlan)$
1. **For** $i = 1$ to $Num\_Feasible$ **Do** //$Num\_Feasible$ is the number of actions feasible in the current state
2. **Begin**
3. **If** $Goal \subset CurState$ then return $CurPlan$
4. **If** $n = 0$ then return FAIL
5. $act := BackwardReg(CurGoal, MaxDeep)$ //this function chooses an action which was not chosen before
6. $NewCurState := apply(CurState, act)$
7. $NewCurPlan := append(CurPlan, act)$
8. $Answer = ComputePlan(NewCurState, n-1, NewCurPlan)$
9. **If** $Answer$ is not FAIL then return $(Answer)$
10. **End**
11. Return FAIL

Fig. 1. The main algorithm

3.2 Auxiliary procedures for the proposed forward search heuristics

In this section, we describe two auxiliary procedures (lines 2 and 4 of the $BackwardReg$ function). The first orders the propositions of a given subgoal. The second finds an action achieving the given proposition and satisfying some additional constraints.

Goal ordering is based on a number of criteria. The main criterion is a modified version of the GRAPHPLAN criterion from [10]. This criterion is mentioned

$BackwardReg(CurGoal, MaxDeep)$
1. If $MaxDeep = 0$ Choose randomly an action feasible in the current step that was not chosen before, and return it
2. Order propositions of $CurGoal$ by order of their achievement
3. Let $CurProp$ be the proposition of the $CurGoal$, which is minimal in $CurGoal \setminus CurState$
4. Choose an action act achieving $CurProp$ that was not chosen before in the $CurState$.
5. If it is not possible to choose such an action, then choose randomly an action feasible in the current step that was not chosen before, and return it
6. If act is feasible in the current state then return act
7. Let $NewCurGoal$ be the set of preconditions of act
8. Return($BackwardReg(NewCurGoal, MaxDeep - 1)$)

Fig. 2. The Main Heuristic of the Algorithm

in 2.2. It states that for two propositions p and q, p must be achieved before q if every action achieving p conflicts with q. The modified version used here states that p must be achieved before q if the percent of actions conflicting with q among actions achieving p is more than the percent of actions conflicting with p among actions achieving q.

Thus, if for two given propositions p and q the "chance"that q will be destroyed while achieving p is higher than the "chance" that p will be destroyed while achieving q, it is preferable to achieve p before q. The original version of this criterion was derived from analysis of problems such as BlocksWorld, HanoiTower and so on, where it is directly applicable. The proposed modification of this method extends its applicability. For example it is applicable for many Logistics-like problem.

Intuitively, the action selection function uses the following rule: Select an action that can be the last action in a plan achieving the required proposition from the current state. In particular, the action selection function performs three steps. First, it computes the **relevant set** which contains the **required** proposition and all propositions that are mutually exclusive with it. The **non-relevant set** is determined as the complement of the relevant set. Next, it builds a **transition graph** whose nodes correspond to the elements of the **relevant** set. This graph contains an edge (a, b) for each action with a precondition a and an add-effect b. Finally, it selects an action corresponding to the last edge in a path from a proposition that is true in the current state to the **required** proposition. If there is no such path, it returns $FAIL$. If there are few such paths, it chooses a path with the minimal number of **non-relevant** preconditions of the corresponding actions (in order to find a path as close as possible to a "real" plan).

3.3 Optimizations

The actual implementation of RO introduces a number of optimizations to the above algorithm. We describe them next.

The first feature is a more complicated backtrack condition. There are basically two conditions that can trigger backtracking: either the plan exceeds some fixed length or a state has been visited twice. However, the first backtrack must be triggered by the first condition. The state we backtrack to is determined as follows: If no state appears more than once (i.e., we backtracked because of plan length), we simply backtrack one step. If a state appears twice in the current state sequence, we backtrack to the state prior to second appearance of the first repeated state. For example, suppose our maximal plan length is 6, the state sequence is (A, C, B, B, C, D), and we have not backtracked before. In this sequence, both B and C appear twice. However, B is the first state to occur twice. Therefore, we backtrack to before the second appearance of B. Thus, the new sequence, after backtracking, is (A, C, B). From this point on, we backtrack whenever the current state appears earlier in the sequence, even if plan length is smaller than the maximal length.

The second optimization is the memoization of the sequence of subgoals generated by the main heuristics. Instead of recomputing the whole subgoal sequence in each application of the search heuristic, we use the subgoal sequence that was constructed in the previous application (if such a sequence exists). The modified algorithm eliminates from the tail of the subgoal sequence all subgoals that were achieved in the past, and continues construction from the resulting sequence.

This memoization method has two advantages. First, it saves time by avoiding the computation of the full subgoal sequence. Second, and more importantly, is that it maintains a connection between subsequent applications of the search heuristics. This way, each application of the search heuristic application builds on top of the results of the previous application and avoids accidental destruction of these results. The running example in the next section demonstrates the usefulness of this approach.

3.4 A Running Example

Consider a well-known instance of the BlocksWorld domain called the Sussman Anomaly. It is an instance with three blocks, its initial state is $\{on(3,1), on-table(1), clear(3), on-table(2), clear(2)\}$ and the goal is $\{on(1,2), on(2,3)\}$. Note, there is only a single reachable state that satisfies the goal criteria. In this state, the proposition $on-table(3)$ holds. However, $on-table(3)$ is not stated explicitly in the goal. This raises a difficulty for algorithms that employ goal ordering because they are strongly affected by interactions between actions in the goal. For example, in our case, the propositions of the goal have to be ordered as follows $(on(2,3), on(1,2))$. However, before achieving $on(2,3)$, it is necessary to achieve $on-table(3)$.

Let us run RO on this instance. We consider the simplest version of the BlocksWorld domain with two actions only: one for moving a block from the table on top of another block, and one for moving a block to the table. Below we show the result of each application of the search heuristics.

First application.

The current state is the initial state, i.e. $\{on(3,1)on-table(1), clear(3), on-table(2), clear(2)\}$, the subgoal sequence is not constructed yet, so it needs to be constructed from scratch. Its first subgoal is the main goal which is ordered as $(on(2,3), on(1,2))$. The required proposition of this level is $on(2,3)$, RO selects action $put-on(2,3)$ to achieve this proposition. This action is feasible in the current state and it is returned, so the first application is finished here. Note, that the first application of the search heuristics selects a wrong action!

Second application. The current state is $\{on(2,3), on(3,1), on-table(1), clear(2)\}$. The only subgoals in the subgoal sequence are $(on(2,3), on(1,2))$. RO chooses the required proposition to be $on(1,2)$ and selects action $put-on(1,2)$ to achieve this proposition. As we can see, this action is not feasible in the current state and so the subgoal sequence is extended. Now, it contains both the main goal and the preconditions of action $put-on(1,2)$, namely, it is $((on(2,3), on(1,2))(clear(1), clear(2)))$. Note, that the ordering of the second subgoal is chosen randomly and it does not matter here. Now, the required proposition is $clear(1)$. The action selection heuristic chooses action $take-out(3,1)$. This action is also not feasible and this fact leads us to further extend the subgoal sequence. The next ordered subgoal in this sequence will be $(on(3,1), clear(3))$ (again, the order of the propositions does not matter here). The new required proposition will be $clear(3)$. To achieve this proposition, RO selects action $take-out(2,3)$, which is returned, because it is feasible in the current state. Note that in spite of the fact that after an application of this action we arrive at a state that appeared before, backtracking is not performed because the first backtrack occurs only once we exceed the maximal plan length – and this did not occur, yet.

Third application. The new current state is the initial state! However, we have learned something in the process, and this is reflected in the sequence of subgoals we now have: $((on(2,3), on(1,2))(clear(1), clear(2)))$. (The last subgoal was eliminated because it was fully achieved). This information was not available when we started our search. In fact, the new required proposition is $clear(1)$, a proposition that does not appear in the original goal. Because we have chosen it as the required proposition, we will not repeat past mistakes. The algorithm selects action $take-out(3,1)$ to achieve this proposition. This action is feasible in the current state, therefore, it is returned.

During the fourth and the fifth application, the algorithm achieves the main goal in a straightforward way: it puts block 2 on block 3 and then block 1 on block 2.

The resulting plan is : $(put-on(2,3), take-out(2,3), take-out(3,1), put-on(2,3), put-on(1,2))$

Obviously, this plan is not optimal. The first two applications were spent constructing a subgoal sequence which then forced RO to select the right actions. This feature of RO frequently leads to non-optimal plans. However, we believe that the resulting non-optimal plan usually contains a near-optimal plan as a subsequence. Therefore, we can run a plan refinement algorithm [1] on the output of RO and obtain a near optimal plan. In some cases, this may be a more effective approach for obtaining a near optimal plan.

4 Experimental Analysis

To determine the effectiveness of RO, we performed a number of experiments comparing its performance to the FF planner – the winner of the AIPS 2000 planning competition. These results are described and analyzed in this section.

All experiments were conducted on a SUN Ultra4 with 1.1GB RAM. Each result is given in the form A/B where A is the running time for the given instance, and B is the length of the returned plan. The input language is a restricted version of PDDL without conditional effects.

The main conclusion of our experimental analysis is that RO is competitive with FF, outperforming it on domains in which the subproblems obtained after ordering the top level goal are non-trivial.

4.1 Classical Domains

In this subsection we consider well known classical domains, such as the BlocksWorld, the Hanoi Tower, and two versions of the Logistics. The results are presented in the table below.

BlocksWorld			Hanoi Tower		
size	RO	FF	size	RO	FF
10	0.4/12	0.08/12	6	0.2/63	0.12/63
15	2/18	0.14/17	7	0.3/127	0.3/127
20	7/27	0.4/26	8	0.6/255	1.3/255
25	19.9/36	1.01/35	9	1.3/511	3.61/511
30	46.5/44	2.64/44	10	2.9/1023	23.06/1023
Usual Logistics			Logistics With Car Transportation		
size	RO	FF	size	RO	FF
10	0.8/105	0.65/95	10	0.7/109	0.47/80
20	8.5/210	10.5/191	20	7.3/239	8.83/165
30	63.4/287	100/312	30	31.2/349	57.29/250
40	127.9/419	248.3/383	40	92.7/479	234.17/335
50	314.3/522	806.3/479	50	226.3/583	813/420

Table 1. BlocksWorld Running Results

We can see that RO is not competitive with FF on the BlocksWorld. This stems from the simple nature of the subproblems obtained after goal ordering in

this domain which make the additional computational effort of RO redundant in this case.

However, for some problems harder than BlocksWorld, this computational effort is worthwhile. One such example is the HanoiTower domain. On this domain, FF outperforms RO for small problem sizes (less than 7 discs). But when the number of discs is larger than 7, RO outperforms FF, with the difference increasing as the domain size increases.

The last example in this part is the Logistics domain. We consider two versions of this problem. The first one is the classic domain. The second one is a slight modification of the first, where airplanes can load and unload cars.

An instance of the Logistics is mainly characterized by the initial and final distributions of packages among cities. If the number of cities is small relative to the number of packages or if the majority of packages have the same initial and final locations, FF outperforms RO. However, when packages are distributed among many cities and their final locations are also different, RO outperforms FF. Table 1 contains running times for both version on the Logistics domain. In all the examples we used, each city contained exactly one package and the initial and final location of each package was different. In addition, each instance of the second version contains a single car only.

4.2 Modified Classical Domains

In addition to classical domains, we considered two novel domains which combine features of existing domains.

Combination of the Logistics with the BlocksWorld The first such domain combines aspects of the Logistics and BlocksWorld domains. Suppose we have n locations and m objects placed in these locations. A proposition $at(i,k)$ means that object i is at location k. If object i can move from location l to location k, this fact is expressed as $moves(i,k,l)$. We assume the graph of moves to be undirected for each object, that is, $moves(i,k,l)$ implies $moves(i,l,k)$. Objects can transport each other. Propositions $transports(i,k)$ and $in(i,k)$ mean that the object i can transport the object k and that the object i is within the object k respectively. For this domain, we assume that the transport graph is a DAG. The transport graph is defined as follows: the nodes are the objects and an edge (a,b) appears in it iff the object a can transport the object b.

The BlocksWorld features of this domain are expressed by the fact that we can put one object on another. The proposition expressing BlocksWorld-like relations are $clear(i)$, $at(i,k)$ and $on(i,k)$. Note, that $at(i,k)$ means that the object i is "on the table" at the location k. This type of proposition plays the role of the connecting link between these two combined domains.

The set of actions in this domain is the union of the actions in the Logistics and the BlocksWorld domain with a few small modifications. In particular, an object can be loaded into another object or moved between locations only if it is "clear" and "on_ground"; also we can put one object into another only if they are in the same location and not within another object.

This domain has an interesting property that neither the Logistics nor the BlocksWorld have: Top level goals interact with intermediate goals. An object, which is in intermediate level of a tower at some location, may be required for transportation of another object. To do so, we must remove all the objects above this object.

To try planners on this domain, we constructed a simple set of examples, in which the planner have to build towers of blocks in a few different location, and the moving cubes must be in bottom places of these towers. FF behaves badly on this set: it runs more than hour on an example with 11 cubes. However much larger examples of this domain are tractable for RO. For example, it orders 21 cubes in 50 seconds and produces plan of length 628 steps.

A Modified Hanoi Tower A second domain we consider is a modification of the Hanoi Tower domain. In this modified version the number of locations is arbitrary. Initially all discs are in the first location. The task is to move them to the last location using a number of rules. These rules are almost the same as the Hanoi Tower domain rules with two exceptions: The first one is that if a disc is placed on the final location it can't be taken back from this location. The second one is that all discs are enumerated and it is possible to put disc number a on disc number b iff $b = a + 1$ or $b = a + 2$. In essence, this domain is a simplified form of the FreeCell domain.

The difficulty of an instance in this domain depends on two factors: the number of discs and the number of cells. The latter determines the constrainedness of the instance (the fewer the cells, the more constrained the instance is).

For small number of discs (less than 12) FF outperforms RO independently of constrainedness of the processed instance. This is the case for weakly constrained instances with large number of discs, as well. However, tightly constrained instances of this domain are practically intractable for FF. The table below presents running results of RO for instances whose solution for FF takes more than one and a half hours. The size of an instance is given in form A/B, where A is the number of discs, B is the number of locations except for the final one.

instance	time/length
17/5	527.3/933
18/6	76/279
20/7	52/160
22/7	62/185
24/8	120/265
25/8	152/287
26/9	155/243
28/9	269/207
30/9	550/303

Table 2. Running times for the Modified Hanoi Tower domain

5 Conclusions

In this paper we presented a forward search planning algorithm. An implementation of this algorithm was shown to be competitive with FF on domains in which subproblems obtained as a result of goal ordering are themselves non-trivial. Our algorithm makes a number of novel contributions:

- A forward search heuristics combining backward regression and goal ordering techniques.
- A complex memoization technique for reusing subgoal sequences.
- A novel combination of the Logistics and the BlocksWorld domain.
- A better understanding of the weaknesses and strengths of FF.

References

1. F. Bacchus,Y. Teb *Making Forward Chaining Relevant*, AIPS-98, pages 54-61, 1998
2. A. Blum, M. Furst *Fast Planning Through Planning Graph Analysis*, Artificial Intelligence, 90(1997), pages 281-300, 1997.
3. B. Bonet, H. Geffner. *Planning as Heuristic Search: New Results*, Artificial Intelligence, Proceedings of the 5th European Conference on Planning, pages 359-371, 1999.
4. R.I.Brafman. *Reachability, Relevance, Resolution and the Planning as Satisfiability Approach*,In Proceedings of the IJCAI' 99, 1999.
5. P. Haslum, H. Geffner. *Admissible Heuristics for Optimal Planning*, AIPS2000 pages 140-149, 2000.
6. J. Hoffman, B. Nebel. *The FF Planning System:Fast Plan Generation Through Extraction of Subproblems*, to appear in JAIR.
7. J. Hullen, F. Weberskirch. *Extracting Goal orderings to Improve Partial-Order Planning*, PuK99, pages 130-144, 1999.
8. S. Kambhampati,R. Nigenda. *Distance-based Goal-ordering Heuristics for Graphplan*, AIPS2000 pages 315-322, 2000.
9. S. Kambhampati,E. Parker,E. Lambrecht. *Understanding and Extending Graphplan*, 4th European Conference of Planning, pages 260-272, 1997.
10. J. Koehler, J. Hoffman. *On Reasonable and Forced Goal Ordering and their Use in an Agenda-Driven Planning Algorithm*, JAIR 12(2000), pages 339-386.
11. J. Koehler, B. Nebel, J. Hoffman, Y. Dimopoulos. *Extending Planning Graphs to ADL Subset*, ECP97, pages 273-285, 1997.
12. R. E. Korf. *Macro-Operators: A Weak Method for Learning*,Aritficial Intelligence, 26 (1985), pages 35-77.
13. D. McDermott. *Using regression-match graphs to control search in planning.*, Artificial Intelligence, 109(1-2), pages 111-159, 1999.
14. J.Porteous,L.Sebastia. *Extracting and Ordering Landmarks for Planning*, Technical Report, Dept. of Computer Science, University of Durham, September 2000.
15. I. Razgon. *A Forward Search Planning Algorithm with a Goal Ordering Heuristic*, MSc Thesis, Ben-Gurion University, Israel, 2001.
16. D. Smith, M. Peot. *Suspending Recursion in Partial Order Planning*, AIPS96, 191-198, 1996.
17. D. Weld. *An Introduction to Least Commitment Planning*, AI Magazine 15(4), pages 27-61, 1994.

On the Extraction, Ordering, and Usage of Landmarks in Planning

Julie Porteous[1], Laura Sebastia[2], and Jörg Hoffmann[3]

[1] Department of Computer Science, The University of Durham, Durham, UK, J.M.Porteous@durham.ac.uk

[2] Dpto. Sist. Informáticos y Computación, Universidad Politécnica de Valencia, Valencia, Spain, lstarin@dsic.upv.es

[3] Institut für Informatik, Universität Freiburg, Freiburg, Germany, hoffmann@informatik.uni-freiburg.de

Abstract. Many known planning tasks have inherent constraints concerning the best order in which to achieve the goals. A number of research efforts have been made to detect such constraints and use them for guiding search, in the hope to speed up the planning process.
We go beyond the previous approaches by defining ordering constraints not only over the (top level) goals, but also over the sub-goals that will arise during planning. Landmarks are facts that must be true at some point in every valid solution plan. We show how such landmarks can be found, how their inherent ordering constraints can be approximated, and how this information can be used to decompose a given planning task into several smaller sub-tasks. Our methodology is completely domain- and planner-independent. The implementation demonstrates that the approach can yield significant performance improvements in both heuristic forward search and GRAPHPLAN-style planning.

1 Introduction

Given the inherent complexity of the general planning problem it is clearly important to develop good heuristic strategies for both managing and navigating the search space involved in solving a particular planning instance. One way in which search can be informed is by providing hints concerning the order in which planning goals should be addressed. This can make a significant difference to search efficiency by helping to focus the planner on a progressive path towards a solution. Work in this area includes that of GAM [7] and PRECEDE [9]. Koehler and Hoffmann [7] introduce the notion of *reasonable orders* where a pair of goals A and B can be ordered so that B is achieved before A if it isn't possible to reach a state in which A and B are both true, from a state in which just A is true, without having to temporarily destroy A. In such a situation it is reasonable to achieve B before A to avoid unnecessary effort.

The motivation of the work discussed in this paper is to extend those previous ideas on orderings by not only ordering the (top level) goals, but also the sub-goals that will arise during planning, i.e., by also taking into account what we call the *landmarks*. The key feature of a landmark is that it *must* be true on any solution path to the given planning task. Consider the *Blocksworld* task shown in Figure 1, which will be our working example throughout the paper.

Here, *clear(C)* is a landmark because it will need to be achieved in any solution plan. Immediately stacking B on D from the initial state will achieve one of the top level goals of the task but it will result in wasted effort if *clear(C)* is not achieved first. The ordering *clear(C)* $\leq$ *on(B D)* is, however, *not* reasonable in terms of Koehler and Hoffmann's definition yet it is a sensible order to impose if we wish

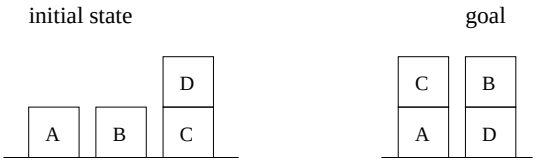


Fig. 1. Example *Blocksworld* task.

to reduce wasted effort during plan generation. We introduce the notion of *weakly reasonable* orderings, which captures this situation. Two landmarks L and L' are also often ordered in the sense that all valid solution plans make L true before they make L' true. We call such ordering relations *natural*. For example, *clear(C)* is naturally ordered before *holding(C)* in the above *Blocksworld* task.

We introduce techniques for extracting landmarks to a given planning task, and for approximating natural and weakly reasonable orderings between those landmarks. The resulting information can be viewed as a tree structure, which we call the *landmark generation tree*. This tree can be used to decompose the planning task into small chunks. We propose a method that does not depend on any particular planning framework. To demonstrate the usefulness of the approach, we have used the technique for control of both the forward planner FF(v1.0) [5] and the GRAPHPLAN-style planner IPP(v4.0) [8], yielding significant performance improvements in both cases.

The paper is organised as follows. Sections 2 to 4 explain how landmarks can be extracted, ordered, and used, respectively. Empirical results are discussed in Section 5 and we conclude in Section 6.

2 Extracting Landmarks

Throughout the paper, we consider a propositional STRIPS framework where actions are triples $o = (\mathrm{pre}(o), \mathrm{add}(o), \mathrm{del}(o))$, plans are sequences of actions, the result of applying an action o to a state S with $\mathrm{pre}(o) \subseteq S$ is $Result(S, o) = (S \cup \mathrm{add}(o)) \setminus \mathrm{del}(o)$, and planning tasks are triples $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ comprising the action set, the initial state, and the goal state. In this section, we will focus on the landmarks extraction process and its properties. First of all, we define what a landmark is.

Definition 1. *Given a planning task $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$. A fact L is a* landmark *in $\mathcal{P}$ iff L is true at some point in all solution plans, i.e., iff for all $P = \langle o_1, \ldots, o_n \rangle, \mathcal{G} \subseteq Result(\mathcal{I}, P) : L \in Result(\mathcal{I}, \langle o_1, \ldots, o_i \rangle)$ for some $0 \leq i \leq n$.*

All initial facts are trivially landmarks (let $i = 0$ in the above definition). For the final search control, they are not considered. They *can*, however, play an important role for extracting ordering information. In the *Blocksworld* task shown in Figure 1, *clear(C)* is a landmark, but *on(A B)*, for example, is not. In general, it is PSPACE-hard to decide whether an arbitrary fact is a landmark.

Definition 2. *Let* LANDMARK RECOGNITION *denote the following problem.*
Given a planning task $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$, and a fact L. Is L a landmark in $\mathcal{P}$?

Theorem 1. *Deciding* LANDMARK RECOGNITION *is* PSPACE*-hard.*

Proof Sketch: By a reduction of (the complement of) PLANSAT, the problem of deciding whether an arbitrary STRIPS planning task is solvable [2]: add an artificial by-pass to the task, on which a new fact L must be added. ∎

Due to space restrictions, we include only short proof sketches in this paper. The complete proofs can be found in a technical report [11]. The following is a simple sufficient condition for a fact being a landmark.

Proposition 1. *Given a planning task $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$, and a fact L. Define $\mathcal{P}_L = (\mathcal{O}_L, \mathcal{I}, \mathcal{G})$ as follows.*

$$\mathcal{O}_L := \{(pre(o), add(o), \emptyset) \mid (pre(o), add(o), del(o)) \in \mathcal{O}, L \notin add(o)\}$$

If $\mathcal{P}_L$ is unsolvable, then L is a landmark in $\mathcal{P}$.

With $\mathcal{P}_L$ being unsolvable, the goal can not be reached without adding L, even when ignoring delete lists. Deciding about solvability of planning tasks with empty delete lists can be done in polynomial time by a GRAPHPLAN-style algorithm [1, 6]. An idea is, consequently, to evaluate the above sufficient condition for each non-initial state fact in turn. However, this can be costly when there are many facts in a task. We use the following two-step process.

1. First, a backward chaining process extracts landmark candidates.
2. Then, evaluating Proposition 1 eliminates those candidates that are not provably landmarks.

The backward chaining process can select initial state facts, but does not necessarily select all of them. In verification, initial (and goal) facts need not be considered as they are landmarks by definition.

2.1 Extracting Landmark Candidates

Candidate landmarks are extracted using what we call the *relaxed planning graph* (RPG): relax the planning task by ignoring all delete lists, then build GRAPHPLAN's planning graph, chaining forward from the initial state of the task to a graph level where all goals are reached. Because the delete lists are empty, the graph does not contain any mutex relations [6]. Once the RPG has been built, we step backwards through it to extract what we call the *landmark-generation tree* (LGT). This is a tree (N, E) where the nodes N are candidate landmarks and an edge $(L, L') \in E$ indicates that L must be achieved as a necessary prerequisite for L'. Additionally, if several nodes $L_1, \ldots, L_k$ are ordered before the same node L', then $L_1, \ldots, L_k$ are grouped together in an AND-node in the sense that those facts must be true together at some point during the planning process. The root of the tree is the AND-node representing the top level goals.

The extraction process is straightforward. First, all top level goals are added to the LGT and are posted as goals in the first level where they were added in the RPG. Then, each goal is solved in the RPG starting from the last level. For each goal g in a level, all actions achieving g are grouped into a set and the intersection I of their preconditions is computed. For all facts p in I we: post p as a goal in the first RPG level were it is achieved; insert p as a node into the LGT; insert an edge between p and g into the LGT. When all goals in a level are achieved, we move on to the next lower level. The process stops when the first (initial) level is reached.

We also use the following technique, to obtain a larger number of candidates: when a set of actions solves a goal, we also compute the union of the preconditions that are not in the intersection. We then consider all actions achieving these facts. If the intersection of those action's preconditions is non-empty, we take the facts in that intersection as candidate landmarks as well. More details about the process (which are not necessary for understanding the rest of this discussion) are described by Porteous and Sebastia [10].

L_0	A_1	L_1	A_2	L_2	A_3	L_3
on-table A	pick-up A	holding A	stack B A	on B A	stack C A	on C A
on-table B	pick-up B	holding B	stack B D	on B D	stack C B	on C B
on-table C	unstack D C	holding D	stack B C	on B C	stack C D	on C D
on D C		clear C	put-down B	...	...	...
clear A			...			
clear B			pick-up C	holding C		
clear D			...	...		
arm-empty						

Fig. 2. Summarised RPG for the *Blocksworld* example shown in Figure 1.

Let us illustrate the extraction process with the *Blocksworld* example from Figure 1. The RPG corresponding to this task is shown in Figure 2. As we explained above, the extraction process starts by adding two nodes representing the goals *on(C A)* and *on(B D)* to the LGT ($N = \{$*on(C A)*,*on(B D)*$\}, E = \emptyset$). It also posts *on(C A)* as goal in level 3 and *on(B D)* in level 2. There is only one action achieving *on(C A)* in level 3: *stack C A*. So, *holding(C)* and *clear(A)* are new candidates. *holding(C)* is posted as a goal in level 2, *clear(A)* is initially true and does therefore not need to be posted as a goal. The new LGT is: $N = \{$*on(C A)*,*on(B D)*,*holding(C)*,*clear(A)*$\}$, $E = \{$(*(holding C)*,*on(C A)*), (*(clear A)*,*on(C A)*$\}$. As there are no more goals in level 3, we move downwards to solve the goals in level 2. We now have two goals: *on(B D)* and *holding(C)*. In both cases, there is only one action adding each fact (*stack B D* and *pick-up C*), so their preconditions *holding(B)*, *clear(D)*, *clear(C)*, *on-table(C)*, and *arm-empty()*, as well as the respective edges, are included into the LGT. The goals at level 1 are *holding(B)* and *clear(C)*, which are added by the single actions *pick-up B* and *unstack D C*. The process ends up with the following LGT, where we leave out, for ease of reading, the initial facts and their respective edges: $N = \{$*on(C A)*, *on(B D)*, *holding(C)*, *holding(B)*, *clear(C)*, ... $\}$ and $E = \{$(*holding(C)*,*on(C A)*), (*holding(B)*,*on(B D)*), (*clear(C)*,*holding(C)*),... $\}$. Among the parts of the LGT concerning initial facts, there is the edge (*clear(D)*,*clear(C)*) $\in E$. As we explain in Section 3, this edge plays an essential role for detecting the ordering constraint *clear(C)* $\leq$ *on(B D)* that was mentioned in the introduction. The edge is inserted as precondition of *unstack D C*, which is the first action in the RPG that adds *clear(C)*.

2.2 Verifying Landmark Candidates

Say we want to move from city A to city D on the road map shown in Figure 3, using a standard *move* operator. Landmarks extraction will come up with the following LGT: $N = \{$*at(A)*, *at(E)*, *at(D)*$\}$, $E = \{$(*at(A)*, *at(E)*), (*at(E)*, *at(D)*)$\}$—the RPG is only built until the goals are reached the first time, which happens in this example before *move C D* comes in. However, the action sequence $\langle$*move(A,B)*, *move(B,C)*, *move(C,D)*$\rangle$ achieves the goals without making *at(E)* true. Therefore, the candidate *at(E)* $\in N$ is not really a landmark.

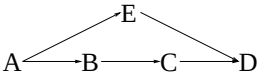


Fig. 3. An example road map.

We want to remove such candidates because they can lead to incompleteness in our search framework, which we will describe in Section 4. As was said above, we simply check Proposition 1 for each fact $L \in N$ except the initial facts and the goals, i.e., for each such L in turn we ignore the actions that add L, and check solvability of

the resulting planning task when assuming that all delete lists are empty. Solvability is checked by constructing the RPG to the task. If the test fails, i.e., if the goals are reachable, then we remove L from the LGT. In the above example, *at(A)* and *at(D)* need not be verified. Ignoring all actions achieving *at(E)*, the goal is still reachable by the actions that *move* to D via B and C. So *at(E)* and its edges are removed, yielding the final LGT where $N = \{at(A), at(D)\}$ and $E = \emptyset$.

3 Ordering Landmarks

In this section we define two types of ordering relations, called *natural* and *weakly reasonable* orders, and explain how they can be approximated. Firstly, consider the natural orderings. As said in the introduction, two landmarks L and L' are ordered naturally, $L \leq_n L'$, if in all solution plans L is true before L' is true. L is true before L' in a plan $\langle o_1, \ldots, o_n \rangle$ if, when i is minimal with $L \in Result(\mathcal{I}, \langle o_1, \ldots, o_i \rangle)$ and j is minimal with $L' \in Result(\mathcal{I}, \langle o_1, \ldots, o_j \rangle)$, then $i < j$. Natural orderings are characteristic of landmarks: usually, the reason why a fact is a landmark is that it is a necessary prerequisite for another landmark. For illustration consider our working example, where *clear(C)* must be true immediately before *holding(C)* in all solution plans. In general, deciding about natural orderings is PSPACE-hard.

Definition 3. *Let* NATURAL ORDERING *denote the following problem.*

Given a planning task $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$, and two atoms A and B. Is there a natural ordering between B and A, i.e., does $B \leq_n A$ hold?

Theorem 2. *Deciding* NATURAL ORDERING *is* PSPACE*-hard.*

Proof Sketch: Reduction of the complement of PLANSAT. Arrange actions for two new facts A and B such that one can either: add A, then B, then solve the original task; or add B, then A, then achieve the goal right away. ∎

The motivation for weakly reasonable orders has already been explained in the context of Figure 1. Stacking B on D from the initial state is not a good idea since $clear(C)$ needs to be achieved first if we are to avoid unnecessary effort. However, the ordering *clear(C)* $\leq$ *on(B D)* is *not* reasonable, in the sense of Koehler and Hoffmann's formal definition [7], since there are reachable states where B is on D and C is not clear, but C can be made clear without unstacking B. However, reaching such a state requires unstacking D from C, and (uselessly) stacking A onto C. Such states are clearly not relevant for the situation at hand. Our definition therefore weakens the reasonable orderings in the sense that only the *nearest* states are considered in which B is on D. Precisely, Koehler and Hoffmann [7] define $S_{A,\neg B}$, for two atoms A and B, as the set of reachable states where A has just been achieved, but B is still false. They order $B \leq_r A$ if all solution plans achieving B from a state in $S_{A,\neg B}$ need to destroy A. In contrast, we restrict the starting states that are considered to $S^{opt}_{A,\neg B}$, defined as those states in $S_{A,\neg B}$ that have minimal distance from the initial state. Accordingly, we define two facts B and A to have a weakly reasonable ordering constraint, $B \leq_w A$, iff

$$\forall\, s \in S^{opt}_{(A,\neg B)} :\ \forall\, P \in \mathcal{O}^* :\ B \in Result(s, P) \Rightarrow \exists\, o \in P : A \in \mathrm{del}(o)$$

Deciding about weakly reasonable orderings is PSPACE-hard.

Definition 4. *Let* WEAKLY REASONABLE ORDERING *denote the following problem.*

Given a planning task $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$, and two atoms A and B. Is there a weakly reasonable ordering between B and A, i.e., does $B \leq_w A$ hold?

Theorem 3. *Deciding* WEAKLY REASONABLE ORDERING *is* PSPACE*-hard.*

Proof Sketch: Reduction of the complement of PLANSAT. Arrange actions for two new facts A and B such that: A is never deleted, and achieved once before the original task can be started; B can be achieved only when the original goal is solved. ∎

3.1 Approximating Natural and Weakly Reasonable Orderings

As an exact decision about either of the above ordering relations is as hard as planning itself, we have used the approximation techniques described in the following. The approximation of $\leq_n$ is called $\leq_{an}$, the approximation of $\leq_w$ is called $\leq_{aw}$. The orders $\leq_{an}$ are extracted directly from the LGT. Recall that for an edge (L, L') in the LGT, we know that L and L' are landmarks and also that L is in the intersection of the preconditions of the actions achieving L' at its lowest appearance in the RPG. We therefore order a pair of landmarks L and L' $L \leq_{an} L'$, if $LGT = (N, E)$, and $(L, L') \in E$.

What about $\leq_{aw}$, the approximations to the weakly reasonable orderings? We are interested in pairs of landmarks L and L', where from all nearby states in which L' is achieved and L is not, we must delete L' in order to achieve L. Our method of approximating this looks at: pairs of landmarks within a particular AND-node of the LGT since these must be made simultaneously true in some state; landmarks that are naturally ordered with respect to one of this pair since these give an ordered sequence in which "earlier" landmarks must be achieved; and any inconsistencies[1] between these "earlier" landmarks and the other landmark at the node of interest. As the first two pieces of information are based on the RPG (from which the LGT is extracted), our approximation is biased towards those states that are close to the initial state. The situation we consider is, for a pair of landmarks in the same AND-node in the LGT, what if a landmark that is ordered before one of them is inconsistent with the other? If they are inconsistent then this means that they can't be made simultaneously true, (ie achieving one of them will result in the other being deleted). So that situation is used to form an order in one of the following two ways:

1. landmarks L and L' in the same AND-node in the LGT can be ordered $L \leq_{aw} L'$, if:

$$\exists\ x \in Landmarks : x \leq_{an} L \wedge inconsistent(x, L')$$

2. a pair of landmarks L and L' can be ordered $L \leq_{aw} L'$ if there exists some other landmark x which is: in the same AND-node in the LGT as L'; and there is an ordered sequence of $\leq_{an}$ orders that order L before x. In this situation, L and L' are ordered, if

$$\exists\ y \in Landmarks : y \leq_{an} L \wedge inconsistent(y, L')$$

In both cases the rationale is: look for an ordered sequence of landmarks required to achieve a landmark x at a node. For any landmark L in the sequence, if L is inconsistent with another landmark L' at the same AND-node as x then there is no point in achieving L' before L (since L' will then be deleted in the effort to achieve x).

A final way in which we derive ordering constraints is based on analysis of any $\leq_{an}$ and $\leq_{aw}$ orders already identified. We ommit the details here and refer the interested reader to our technical report [11].

[1] A pair of facts is inconsistent if they can't be made simultaneously true. We approximate inconsistency using the respective function provided by the TIM API [3] available from: `http://www.dur.ac.uk/computer.science/research/stanstuff/planpage.html`

3.2 Extracting Natural and Weakly Reasonable Orderings

The LGT is used for extracting orders as follows: (i) identify the $\leq_{an}$ orders; (ii) identify the $\leq_{aw}$ orders; (iii) analyse those orders to identify remaining $\leq_{aw}$ orders; (iv) remove cycles in the graph of orders; (v) finally, add all orders as edges in the LGT for later use during planning.

As an illustration of this process, consider again the example shown in figure 1. First, the set of $\leq_{an}$ orders are extracted directly from the LGT. The set contains, amongst other things: *clear D* $\leq_{an}$ *clear C*, *clear(C)* $\leq_{an}$ *holding(C)*, and *holding(C)* $\leq_{an}$ *on(C A)* (see Section 2). In the next step, the $\leq_{aw}$ orders are identified. Let us focus on how the order *clear(C)* $\leq_{aw}$ *on(B D)* (our motivating example) is found. From the $\leq_{an}$ orders we have the ordered sequence $\langle$*clear(D)*, *clear(C)*, *holding(C)*, *on(C A)*$\rangle$ and the fact that *on(C A)* is in the same node as *on(B D)*. Since *clear(D)* and *on(B D)* are inconsistent and *clear(D)* $\leq_{an}$ *clear(C)*, the order *clear(C)* $\leq_{aw}$ *on(B D)* is imposed. Note here the crucial point that we have the order *clear(D)* $\leq_{an}$ *clear(C)*. We have that order because *unstack D C* is the first action in the RPG that adds *clear(C)*. The nearest possibility, from the initial state, of clearing C is to unstack D from C. This can only be done when D is still clear. Our approximation methods recognise this, and correctly conclude that stacking B on D immediately is not a good idea.

The next stage is a check to identify and remove any cycles that appear in the graph of orderings. A cycle (or strongly connected component) such as, $L \leq_{an} L'$ and $L' \leq_{aw} L$, might arise if a landmark must be achieved more than once in a solution plan (for example, in the *Blocksworld* domain this is frequently the case for *arm_empty()*). At present, any cycles in the orders are removed since the search process needs the LGT to be a tree structure. They are removed by firstly identifying for each cycle the set of articulation points for it (a node in a connected component is an articulation point if the component that remains, after the node and all edges incident upon it are removed, is no longer connected). The cycles are broken by iteratively removing the articulation points and all edges incident upon these points until no more strongly connected components remain. For our small example no cycles are present so the final step is to add the $\leq_{aw}$ orders to the LGT.

4 Using Landmarks

Having settled on algorithms for computing the LGT, there is still the question of how to use this information during planning. For use in forward state space planning, Porteous and Sebastia [10] have proposed a method that prunes states where some landmark has been achieved too early. If applying an action achieves a landmark L that is not a leaf of the current LGT, then do not use that action. If an action achieves a landmark L that *is* a leaf, then remove L (and all ordering relations it is part of) from the LGT. In short, do not allow achieving a landmark unless all of its predecessors have been achieved already.

Here, we explore an idea that uses the LGT to decompose a planning task into smaller sub-tasks, which can be handed over to any planning algorithm. The idea is similar to the above described method in terms of how the LGT is looked at: each sub-task results from considering the leaf nodes of the current LGT, and when a sub-task has been processed, then the LGT is updated by removing achieved leaf nodes. The main problem is that the leaf nodes of the LGT can often not be achieved as a conjunction. The main idea is to pose those leaf nodes as a *disjunctive* goal instead. See the algorithm in Figure 4.

The depicted algorithm keeps track of the current state S, the current plan prefix P, and the current disjunctive goal $Disj$, which is always made up out of the current leaf nodes of the LGT. The initial facts are immediately removed because they are

$S := \mathcal{I}$, $P := \langle\,\rangle$
remove from LGT all initial facts and their edges
repeat
  $Disj :=$ leaf nodes of LGT
  call base planner with actions $\mathcal{O}$, initial state S and goal condition $\bigvee Disj$
  if base planner did not find a solution P' **then** fail **endif**
  $P := P \circ P'$, $S :=$ result of executing P' in S
  remove from LGT all $L \in Disj$ with $L \in \mathrm{add}(o)$ for some o in P'
until LGT is empty
call base planner with actions $\mathcal{O}$, initial state S and goal $\bigwedge \mathcal{G}$
if base planner did not find a solution P' **then** fail **endif**
$P := P \circ P'$, output P

Fig. 4. Disjunctive search control algorithm for a planning task $(\mathcal{O}, \mathcal{I}, \mathcal{G})$, repeatedly calling an arbitrary planner on a small sub-task.

true anyway. When the LGT is empty—all landmarks have been processed—then the algorithm stops, and calls the underlying base planner from the current state with the original (top level) goals. The algorithm fails if at some point the planner did not find a solution.

Looking at Figure 4, one might wonder why the top level goals are no sooner given special consideration than when all landmarks have been processed. Remember that all top level goals are also landmarks. An idea is to force the algorithm, once a top level goal G has been achieved, to keep G true throughout the rest of the process. We have experimented with a number of variations of this idea. The problem with this is that one or a set of already achieved original goals might be inconsistent with a leaf landmark. Forcing the achieved goals to be true together with the disjunction yields in this case an unsolvable sub-task, making the control algorithm fail. In contrast to this, we will see below that the simple control algorithm depicted above is completeness preserving under certain conditions fulfilled by many of the current benchmarks. Besides this, keeping the top level goals true did not yield better runtime or solution length behaviour in our experiments. This may be due to the fact that, unless such a goal is inconsistent with some landmark ahead, it is kept true anyway.

4.1 Theoretical Properties

The presented disjunctive search control is obviously planner-independent in the sense that it can be used within any (STRIPS) planning paradigm—a disjunctive goal can be simulated by using an artificial new fact G as the goal, and adding one action for each disjunct L, where the action's precondition is $\{L\}$ and the add list is $\{G\}$ (this was first described by Gazen and Knoblock [4]). The search control is obviously correctness preserving—eventually, the planner is run on the original goal. Likewise obviously, the method is not optimality preserving.

With respect to completeness, matters are a bit more complicated. As it turns out, the approach is completeness preserving on the large majority of the current benchmarks. The reasons for this are that there, no fatally wrong decisions can be made in solving a sub-task, that most facts which have been true once can be made true again, and that natural ordering relations are respected by any solution plan. We need two notations.

1. A *dead end* is a reachable state from which the goals can not be reached anymore [7], a task is dead-end free if there are no dead ends in the state space.
2. A fact L is *recoverable* if, when S is a reachable state with $L \in S$, and S' with $L \notin S'$ is reachable from S, then a state S'' is reachable from S' with $L \in S''$.

Many of the current benchmarks are invertible in the sense that every action o has a counterpart $\overline{o}$ that undoes o's effects [7]. Such tasks are dead-end free, and all facts in such tasks are recoverable. Completeness is preserved under the following circumstances.

Theorem 4. *Given a solvable planning task $(\mathcal{O}, \mathcal{I}, \mathcal{G})$, and an LGT (N, E) where each $L \in N$ is a landmark such that $L \notin \mathcal{I}$. If the task is dead-end free, and for $L' \in N$ it holds that either L' is recoverable, or all orders $L \leq L'$ in the tree are natural, then running any complete planner within the search control defined by Figure 4 will yield a solution.*

Proof Sketch: If search control fails, then the current state S is a dead end. If it is not, an unrecoverable landmark L' is added by the current prefix P ($L' \notin \mathcal{I}$ so it must be added at some point). L' was not a leaf node at the time it was added, so there is a landmark L with $L \leq L'$ that gets added after L' in contradiction. ■

Verifying landmarks with Proposition 1 ensures that all facts in the LGT really are landmarks; the initial facts are removed before search begins. The tasks contained in domains like *Blocksworld*, *Logistics*, *Hanoi* and many others are invertible [7]. Examples of dead-end free domains with only natural orders are *Gripper* and *Tsp*. Examples of dead-end free domains where non-natural orders apply only to recoverable facts are *Miconic*-STRIPS and *Grid*. All those domains (or rather, all tasks in those domains) fulfill the requirements for Theorem 4.

5 Results

We have implemented the extraction, ordering, and usage methods presented in the preceding sections in C, and used the resulting search control mechanism as a framework for the heuristic forward search planner FF-v1.0 [5], and the GRAPHPLAN-based planner IPP4.0 [1, 8]. Our own implementation is based on FF-v1.0, so providing FF with the sub-tasks defined by the LGT, and communicating back the results, is done via function parameters. For controlling IPP, we have implemented a simple interface, where a propositional encoding of each sub-task is specified via two files in the STRIPS subset of PDDL. We have changed the implementation of IPP4.0 to output a results file containing the spent running time, and a sequential solution plan (or a flag saying that no plan has been found). The running times given below have been measured on a Linux workstation running at 500 MHz with 128 MBytes main memory. We cut off test runs after half an hour. If no plan was found within that time, we indicate this by a dash. For IPP, we did not count the overhead for repeatedly creating and reading in the PDDL specifications of propositional sub-tasks—this interface is merely a vehicle that we used for experimental implementation. Instead, we give the running time needed by the search control plus the sum of all times needed for planning after the input files have been read. For FF, the times are simply total running times.

For scalability reasons, we ran our FF and IPP implementations on different testing suits. Due to space restrictions, all results are given in Figure 5. The left part of the table shows running time and solution length for FF-v1.0, FF-v1.0 controlled by our landmarks mechanism (FF-v1.0 + L), and FF-v2.2. The last system FF-v2.2 is Hoffmann and Nebel's successor system to FF-v1.0, which goes beyond the first version in terms of a number of goal ordering techniques, and a complete search mechanism that is invoked in case the planner runs into a dead end [6]. Let us consider the domains in Figure 5 from top to bottom. In the *Blocksworld* tasks taken from the BLACKBOX distribution, FF-v1.0 + L clearly outperforms FF-v1.0. The running time values are also better than those for FF-v2.2. Solution

	FF-v1.0		FF-v1.0 + L		FF-v2.2			IPP		IPP+ L	
task	time	steps	time	steps	time	steps	task	time	steps	time	steps
Blocksworld							*Blocksworld*				
bw-large-a	0.01	12	0.17	16	0.01	14	*bw-large-a*	0.17	12	0.36	16
bw-large-b	1.12	30	0.18	24	0.01	22	*bw-large-b*	11.05	18	0.79	26
bw-large-c	-	-	0.24	38	1.02	44	*bw-large-c*	-	-	3.17	38
bw-large-d	7.03	56	0.31	48	0.78	54	*bw-large-d*	-	-	11.73	54
Grid							*Grid*				
prob01	0.07	14	0.26	16	0.07	14	*prob01*	1.84	14	1.15	14
prob02	0.46	39	0.44	26	0.47	39	*prob02*	30.61	29	5.11	30
prob03	3.01	58	1.30	79	2.96	58	*prob03*	-	-	56.08	79
prob04	2.75	49	1.30	54	2.70	49					
prob05	28.42	149	390.01	161	29.39	145					
Logistics							*Logistics*				
prob-38-0	38.03	223	5.93	285	39.61	223	*log-a*	-	-	1.42	61
prob-39-0	101.37	244	6.22	294	98.26	239	*log-b*	-	-	0.91	45
prob-40-0	69.03	245	7.49	308	31.68	251	*log-c*	-	-	1.54	56
prob-41-0	129.15	255	7.73	320	29.85	248	*log-d*	-	-	6.80	80
Tyreworld							*Tyreworld*				
fixit-1	0.01	19	0.18	19	0.01	19	*fixit-1*	0.20	19	0.23	19
fixit-10	26.87	118	3.01	136	0.71	136	*fixit-2*	18.55	30	0.67	32
fixit-20	-	-	26.24	266	10.16	266					
fixit-30	-	-	157.74	396	46.65	396					
Freecell							*Freecell*				
prob-7-1	11.87	56	2.05	44	4.96	48	*prob-2-1*	8.98	9	0.48	10
prob-7-2	4.18	50	1.99	45	4.58	52	*prob-2-2*	9.73	8	0.52	10
prob-7-3	2.29	43	1.88	46	4.07	42	*prob-2-3*	8.37	9	0.53	11
prob-8-1	19.31	63	(2.17)	-	11.32	60	*prob-2-4*	9.17	8	0.49	10
prob-8-2	9.89	57	2.48	49	35.52	61	*prob-2-5*	8.78	9	0.53	10
prob-8-3	2.64	50	2.28	51	4.16	54	*prob-3-1*	-	-	1.15	21
prob-9-1	145.60	84	3.33	72	9.55	73	*prob-4-1*	-	-	1.92	29
prob-9-2	49.17	64	3.22	60	6.77	59	*prob-5-1*	-	-	3.01	36
prob-9-3	3.29	55	2.95	54	5.53	54	*prob-6-1*	-	-	3.76	45
prob-10-1	21.89	84	(3.48)	-	61.85	87					
prob-10-2	15.70	70	(2.95)	-	8.45	66	*Gripper*				
prob-10-3	7.68	56	3.63	61	9.32	64	*prob01*	0.02	11	0.20	15
prob-11-1	(222.48)	-	(3.78)	-	- (160.91)	-	*prob03*	3.25	23	0.29	31
prob-11-2	(17.76)	-	(3.42)	-	117.62 (5.62)	74	*prob20*	-	-	20.85	167
prob-11-3	(35.13)	-	(4.39)	-	10.52	83					

Fig. 5. Left part: running time until a solution was found, and sequential solution length for FF-v1.0, FF-v1.0 with landmarks control (FF-v1.0 + L), and FF-v2.2. Times in brackets specify the running time after which a planner failed because search ended up in a dead end. Right part: running time until a solution was found, and sequential solution length for IPP and IPP with landmarks control (IPP + L). All times are given in seconds.

lengths show some variance, making it hard to draw conclusions. In the *Grid* examples used in the AIPS-1998 competition, running time with landmarks control is better than that of both FF versions on the first four tasks. In *prob05*, however, the controlled version takes much longer time, so it seems that the behaviour of our technique depends on the individual structure of tasks in the *Grid* domain. Solution length performance is again somewhat varied, with a tendency to be longer when using landmarks. In *Logistics*, where we look at some of the largest examples from the AIPS-2000 competition, the results are unmistakable: the control mechanism dramatically improves runtime performance, but degrades solution length performance. The increase in solution length is due to unnecessarily many airplane moves: once the packages have arrived at the nearest airports, they are transported to their destination airports one by one (we outline below an approach how this can be overcome). In the *Tyreworld*, where an increasing number of tyres need to be replaced, runtime performance of FF-v1.0 improves dramatically when using landmarks. FF-v2.2, however, is still superior in terms of running time. In terms of solution lengths our method and FF-v2.2 behave equally, i.e., slightly worse than FF-v1.0.

We have obtained especially interesting results in the *Freecell* domain. Data is given for some of the larger examples used in the AIPS-2000 competition. In *Freecell*, tasks can contain dead ends. Like our landmarks control, the FF search

mechanism is incomplete in the presence of such dead ends [5, 6]. When FF-v1.0 or our enhanced version encounter a dead end, they simply stop without finding a plan. When FF-v2.2 encounters a dead end, it invokes a complete heuristic search engine that tries to solve the task from scratch [6]. This is why FF-v2.2 can solve *prob-11-2*. For all planners, if they encountered a dead end, then we specify in brackets the running time after which they did so. The following observations can be made: on the tasks that FF-v1.0 + L can solve, it is much faster than both uncontrolled FF versions; with landmarks, some more trials run into dead ends, but this happens very fast, so that one could invoke a complete search engine without wasting much time; finally, solution length with landmarks control is in most cases better than without.

The right part of Figure 5 shows the data that we have obtained by running IPP against a version controlled by our landmarks algorithm. IPP normally finds plans that are guaranteed to be optimal in terms of the number of parallel time steps. Using our landmarks control, there is no such optimality guarantee. As a measure of solution quality we show, like in the previous figure, the number of actions in the plans found. Quite obviously, our landmarks control mechanism speeds IPP up by some orders of magnitude across all listed domains. In the *Blocksworld*, solutions appear to get slightly longer. In *Grid*, solution length differs only by one more action used in *prob02*. Running IPP + L on the larger examples *prob04* and *prob05* failed due to a parse error, i.e., IPP's parsing routine failed when reading in one of the sub-tasks specified by our landmarks control algorithm. This is probably because IPP's parsing routine is not intended to read in propositional encodings of planning tasks, which are of course much larger than the uninstantiated encodings that are usually used. So this failure is due to the preliminary implementation that we used for experimentation. In the *Logistics* examples from the BLACKBOX distribution, the solutions contain—like we observed for FF in the experiments described above—unnecessarily many airplane moves; those tasks were, however, previously unsolvable for IPP. In one long testing run, IPP + L solved even the comparatively large *Logistics* task *prob-38-0* (used for the FF variants) within 6571 seconds, finding a plan with 251 steps. In the *Tyreworld*, there is a small increase in solution length to be observed (probably the same increase that we observed in our experiments with FF). Running *fixit-3* failed due to a parse error similar to the one described above for the larger *Grid* tasks. In *Freecell*, where we show some of the smaller tasks from the AIPS-2000 competition, the plans found by IPP + L are only slightly longer than IPP's ones for those few small tasks that IPP can solve. Running IPP + L on any task larger than *prob-6-1* produced parse errors. In *Gripper*, the control algorithm comes down to transporting the balls one by one, which is why IPP + L can solve even the largest task *prob-20* from the AIPS-1998 competition, but returns unnecessarily long plans.

In *Gripper*, and partly also in *Logistics*, the disjunctive search control from Figure 4 results in a trivialisation of the planning task, where goals are simply achieved one by one. While this speeds up the planning process, the usefulness of the found solutions is questionable. The problem is there that our approximate LGT does not capture the structure of the tasks well enough—some goals (like a ball being in room B in *Gripper*) become leaf nodes of the LGT though there are other subgoals which should be cared for first (like some other ball being picked up in *Gripper*). One way around this is trying to improve on the information that is provided by the LGT (we will say a few words on this in Section 6). Another way is to change the search strategy: instead of posing all leaf nodes to the planner as a disjunctive goal, we tried posing a disjunction of *maximal consistent subsets* of those leaf nodes (we approximated consistency of a fact set as pairwise consistency according to the TIM API). In *Gripper*, FF and IPP with landmarks control find the optimal solutions with that strategy, in *Logistics*, the solutions are similar to

those found without landmarks control. This result is of course obtained at the cost of higher running times than with the fully disjunctive method. What's more, posing maximal consistent subsets as goals can lead to incompleteness when an inconsistency remains undetected.

6 Conclusion and Outlook

We have presented a way of extracting and using information on ordered landmarks in STRIPS planning. The approach is independent of the planning framework one wants to use, and maintains completeness under circumstances fulfilled by many of the current benchmarks. Our results on a range of domains show that significant, sometimes dramatic, runtime improvements can be achieved for heuristic forward search as well as GRAPHPLAN-style planners, as exemplified by the systems FF and IPP. The approach does not maintain optimality, and empirically the improvement in runtime behaviour is sometimes (like in *Logistics*) obtained at the cost of worse solution length behaviour. There are however (like in *Freecell* for FF) also cases where our technique improves solution length behaviour.

Possible future work includes the following topics: firstly, one can try to improve on the landmarks and orderings information, for example by taking into account the different "roles" that a top level goal can play (i.e. as a top level goal, or as a landmark for some other goal), or by a more informed treatment of cycles. Secondly, post-processing procedures for improving solution length in cases like *Logistics* might be useful for getting better plans after finding a first plan quickly. Finally, we want to extend our methodology so that it can handle conditional effects.

References

1. Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):279–298, 1997.
2. Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.
3. Maria Fox and Derek Long. The automatic inference of state invariants in tim. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.
4. B. Cenk Gazen and Craig Knoblock. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In Steel and Alami [12], pages 221–233.
5. Jörg Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proc. ISMIS-00*, pages 216–227. Springer-Verlag, October 2000.
6. Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
7. Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12:338–386, 2000.
8. Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In Steel and Alami [12], pages 273–285.
9. T. L. McCluskey and J. M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95, 1997.
10. Julie Porteous and Laura Sebastia. Extracting and ordering landmarks for planning. In *Proc. SIG-00*, 2000.
11. Julie Porteous, Laura Sebastia, and Jörg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. Technical Report 4/01, Department of Computer Science, University of Durham, Durham, England, May 2001. Available from `http://www.dur.ac.uk/ dcs0www/research/stanstuff/planpage.html`
12. S. Steel and R. Alami, editors. *Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, Toulouse, France, September 1997. Springer-Verlag.

The Operational Traffic Control Problem: Computational Complexity and Solutions

Wolfgang Hatzack and Bernhard Nebel

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee, Geb. 52, D-79110 Freiburg, Germany
E-mail: ⟨*last name*⟩@informatik.uni-freiburg.de

Abstract. The operational traffic control problem comes up in a number of different contexts. It involves the coordinated movement of a set of vehicles and has by and large the flavor of a scheduling problem. In trying to apply scheduling techniques to the problem, one notes that this is a job-shop scheduling problem with blocking, a type of scheduling problem that is quite unusual. In particular, we will highlight a condition necessary to guarantee that job-shop schedules can be executed in the presences of the blocking constraint. Based on the insight that the traffic problem is a scheduling problem, we can derive the computational complexity of the operational traffic control problem and can design some algorithms to deal with this problem. In particular, we will specify a very simple method that works well in fast-time simulation contexts.

1 Introduction

Assume a set of vehicles (or physical agents) with starting places, starting times, and (perhaps multiple, sequential) goal locations. The problem is now to move the vehicles as fast as possible to the respective goal locations. This is a problem one encounters when trains in a railway system have to be coordinated, when airplanes have to be coordinated in the air or on the ground, when autonomously guided vehicles (AGVs) in a factory or warehouse have to be coordinated, or when a multi-robot group coordinates the movement of the single robots. Interestingly, the problem does not come up in traditional AI planning domains such as *Logistics* (or more generally *transportation* domains [3]). In these domains we never assume that there are capacity restrictions for locations, which implies that vehicles never interfere with each other when moving around.

In all traffic control problems, we can distinguish between the *strategic*, the *tactic*, and the *operational* level. These levels refer to the time span of a day, a few hours, and a few minutes, respectively. We are mainly interested in how to solve the short-time problem, which is, of course, an *on-line* problem in the sense that we do not know the complete input before we start to solve the problem. However, we will consider only the static variant of the problem in the sequel.

In order to solve the problem, we will make some simplifying assumptions. This will help us in finding a satisfycing solution in acceptable time and will at the same time provide us with enough flexibility in the solution that will allow to accommodate new information.

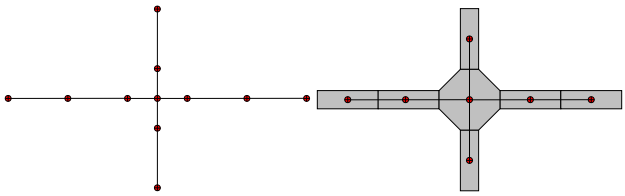
The main simplification we consider is that we assume that the road map for the movements of the vehicles has been fixed in advance. In general, one may want to find solutions independently of a road map. However, this problem can be computationally very demanding. If we are operating in a two-dimensional, rectangular environment and want to coordinate the movement of two-dimensional objects, the decision of whether a goal configuration can be reached is PSPACE-complete [4].

Assuming that the road map is fixed simplifies the problem considerably. However, the problem of finding the minimal number of steps one needs to move all vehicles to the goal locations is still NP-hard as witnessed by the generalized 15-puzzle [10]. For this reason, we will simplify this problem even more. We will assume that the routes the vehicles take are pre-planned and that we only have to *schedule* the movements along these routes. Although this restriction sounds very severe, it is often used in traffic contexts. Furthermore, although simplifying the problem even more, it is still NP-hard to find an optimal solution (as we show below). The resulting problem is similar to the multi-robot *path-coordination* problem [5, p. 379].

The rest of the paper is structured as follows. In Section 2, we formalize the traffic control problem. We then introduce in Section 3 terminology and notions from *scheduling* and show that the traffic problem is a *job-shop* scheduling problem with a *blocking* constraint, which implies that the problem is NP-hard. In Section 4, we will have a look at conditions that guarantee the existence of a solution, and in Section 5 we have a look at methods that allow for "fast-time" simulations. Finally, in Section 6, we report on some experimental data using those methods.

2 The Traffic Control Problem

Each traffic system is based on a specific *infrastructure* that provides facilities on which traffic movements take place, for example roads, airways or waterways. An infrastructure is often represented as a simple graph $G = (V, E)$, where V is a set of intersections or way-points, and E a set of legs. In this paper, however, we represent an infrastructure as a graph where the nodes correspond to a set of resources $R = \{r_1, \ldots, r_m\}$. For our purpose, this allows for a more adequate modeling of infrastructural elements such as intersections, as depicted in Figure 1.



(a) Graph representation (b) Resource representation

Fig. 1. Different representations of an intersection

With $V = \{v_1, \ldots, v_n\}$ we denote a fleet of vehicles that move along the resources of a given infrastructure, where each $v_i \in V$ is associated with a *start time* t_i and an

arbitrary but fixed *route* $\rho_i = (\rho_{i,1} \ldots, \rho_{i,k_i}) \in R^{k_i}$ that might be the result of a path search in the infrastructure or retrieved from a route library, for example. The minimum time it takes v_i to travel along resource $\rho_{i,j}$ is denoted by $\tau_{i,j}$. A *traffic problem* $P = (R, V)$ is a set R of resources and a set V of vehicles with their associated start times and routes. If vehicles never leave the infrastructure, P is called a *closed* traffic problem, whereas in an *open* traffic problem vehicles enter and leave the infrastructure.

The act of v_i moving along resource $\rho_{i,j}$ is called *movement activity* $a_{i,j}$, which allows us to model the movement of all $v_i \in V$ as a *movement plan* $[a_{i,1}, \ldots, a_{i,k_i}]$. A traffic flow arises when vehicles move from their start position at their assigned start time and travel along their specified route to their final position. Formally, a traffic flow F is a set of movement activities $a_{i,j}$ to which a time interval $[\sigma_{i,j}, \phi_{i,j}]$ has been assigned, written $\langle v_i, \rho_{i,j}, [\sigma_{i,j}, \phi_{i,j}]\rangle$. For an orderly movement of vehicles, the following conditions have to be satisfied:

$$\sigma_{i,1} \geq t_i \tag{1}$$

$$\phi_{i,j} - \sigma_{i,j} \geq \tau_{i,j} \tag{2}$$

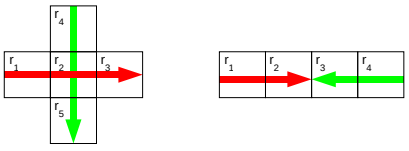
$$\phi_{i,j} = \sigma_{i,j+1}, j \in \{1, \ldots, k_i - 1\} \tag{3}$$

These condition assure that the movement of $v_i \in V$ does not start before its assigned start time t_i, that the actual travel time $\phi_{i,j} - \sigma_{i,j}$ on resource $\rho_{i,j}$ does not fall short of the minimum travel time $\tau_{i,j}$, and finally that the given order of movement activities is preserved without a temporal gap.

For example, if $P = (R, V)$ is a traffic problem with, $V = \{v_1, v_2\}$, $R = \{r_1, r_2, r_3, r_4, r_5\}$, and the connections between the resources as shown in Figure 2 (a), then

$$F = \left\{ \begin{array}{l} \langle v_1, r_1, [0, 10]\rangle, \langle v_1, r_2, [10, 20]\rangle, \langle v_1, r_3, [20, 30]\rangle, \\ \langle v_2, r_4, [0, 10]\rangle, \langle v_2, r_2, [10, 20]\rangle, \langle v_2, r_5, [20, 30]\rangle \end{array} \right\}$$

is a traffic flow for P that satisfies conditions (1) - (3), illustrated in Figure 2 (a).



(a) (b)

Fig. 2. Illustration of traffic flows

If we take a closer look at F, we can see that vehicles v_1 and v_2 plan to use the same resource r_2 simultaneously. Such conflicts can be excluded with the following condition:

$$\rho_{i,j} = \rho_{r,s} \Rightarrow a_{i,j} = a_{r,s} \vee [\sigma_{i,j}, \phi_{i,j}] \cap [\sigma_{r,s}, \phi_{r,s}] = \emptyset \tag{4}$$

However, there is another type of conflict that is an artifact of our movement model. Consider the infrastructure as in Figure 2 (b) and the following traffic flow:

$$F = \left\{ \begin{array}{l} \langle v_1, r_1, [0,10] \rangle, \langle v_1, r_2, [10,20] \rangle, \langle v_1, r_3, [20,30] \rangle, \langle v_1, r_4, [30,40] \rangle, \\ \langle v_2, r_4, [0,10] \rangle, \langle v_2, r_3, [10,20] \rangle, \langle v_2, r_2, [20,30] \rangle, \langle v_2, r_1, [30,40] \rangle \end{array} \right\}$$

Although this traffic flow satisfies conditions (1) - (4), both vehicles are going to exchange their positions at resources r_2 and r_3, which obviously leads to a frontal collision, as sketched in Figure 2 (b). Such situations are avoided if the following condition is satisfied:

$$\rho_{i,j} = \rho_{r,s+1} \wedge \rho_{i,j+1} = \rho_{r,s} \Rightarrow \phi_{i,j} \neq \sigma_{r,s+1} \tag{5}$$

We say that a traffic flow is *safe* if it satisfies conditions (1) - (5). In general, safety has to be established by explicitely resolving conflicts, either by delaying vehicles or assigning new routes to them.[1] Such intervention can be done by human controllers or automatically applying rule-based conflict resolution strategies, for example.

Conflict resolution affects the efficiency of traffic flows, e.g., when a vehicle has to stop in front of an intersection in order to give way to another one. There is a variety of criteria for assessing efficiency of traffic flows. From an infrastructural point of view, an optimal utilization of available resources is desirable, which, for a given set of vehicles, can be achieved by minimizing the latest time at which a vehicle completes its movement. From a vehicle point of view, the most efficient traffic flow minimizes the delay accumulated on the way from its start to finish position.

The completion time C_i and delay D_i of vehicle v_i are defined as follows:

- $C_i = \phi_{i,k_i}$,
- $D_i = \sum_{j=1}^{k_i} \left[(\phi_{i,j} - \sigma_{i,j}) - \tau_{i,j} \right]$.

Based on these definitions, the *maximum completion* time $C_{\max}$ and the *total delay* TD can be defined:

- $C_{\max} = \max_{1 \leq i \leq n} C_i$,
- $TD = \sum_{i=1}^{n} D_i$.

A traffic flow F is *optimal* for TD or $C_{\max}$, if it minimizes the given optimality criterion.

3 Scheduling the Movements: Job-Shop Scheduling with Blocking

Scheduling is concerned with the optimal allocation of scarce resources to activities over time [6]. As we will see in this section, there is a close analogy between finding a safe and optimal traffic flow for a given traffic problem and finding a feasible and optimal schedule for a certain type of scheduling problem.

A scheduling problem $P = (M, J)$ is a set of machines $M = (\mu_1, \ldots, \mu_m)$ and a set of jobs $J = (j_1, \ldots, j_n)$ that have to be processed on machines in M.[2] Typically,

[1] Since we assume routes to be fixed, we do not consider the possibility of re-routing vehicles.

[2] For a general introduction to scheduling, there exists a number of textbooks [1, 2, 8].

scheduling problems are classified in terms of a classification scheme $\{\alpha|\beta|\gamma\}$ [9]. The first field $\alpha = \alpha_1\alpha_2$ describes the machine environment. If $\alpha_1 = O$, we have an *open shop* in which each job j_i consists of a set of operations $\{o_{i,1}, \ldots, o_{i,k_i}\}$ where $o_{i,j}$ has to be processed on machine μ_j for $p_{i,j}$ time units, but the order in which the operations are executed is irrelevant. If $\alpha_1 \in \{F, J\}$, an ordering is imposed on the set of operations corresponding to each job. If $\alpha_1 = F$, we have a *flow shop*, in which each j_i consists of a sequence of m operations $(o_{i,1}, \ldots, o_{i,m})$ and $o_{i,j}$ has to be processed on μ_j for $p_{i,j}$ time units. If $\alpha_1 = J$, we have a *job shop*, in which each each j_i consists of a sequence of k_i operations $(o_{i,1}, \ldots, o_{i,k_i})$ and $o_{i,j}$ has to be processed on a machine $\mu_{i,j} \in M$ for $p_{i,j}$ time units, with $\mu_{i,j} \neq \mu_{i,j+1}$ for i $= 1, \ldots, k_{j-1}$. Note that in a flow shop the *machine routing* is for all jobs the same, while in a job shop the routing is arbitrary but fixed. With α_2 the number of machines can be specified. The second field β indicates a number of job characteristics, for example

- $\{\text{pmtn}\} \subseteq \beta$ (preemption): job splitting is allowed, i.e., the processing of any operation may be interrupted and resumed at a later time.
- $\{\text{nowait}\} \subseteq \beta$: a job must leave a machine immediately after processing is completed. For example, this restriction can be found in the domain of steel production, where molten steel expeditiously has to undergo a series of operations while it has a certain temperature.
- $\{\text{block}\} \subseteq \beta$: a job has to remain on a machine after processing if the next machine is busy. During that time, no other job can be processed on that machine. For example, this phenomena occurs in domains without (or limited) intermediate buffer storage.

The last field γ refers to an optimality criterion which has to be minimized and is a function based on the completion times of jobs which in turn depends on the schedule. The *completion time* of operation $o_{i,j}$ is denoted by $C_{i,j}$ and the time job j_i exits the system is denoted by C_i. Sometimes, for each job j_i a *release date* r_i and a *due date* d_i is specified on which j_i becomes available for processing or should be completed, respectively. With this, the lateness of job j_i can be defined as $L_i = C_i - d_i$ and the *unit penalty* as

$$U_i = \begin{cases} 1 \text{ if } C_i > d_i \\ 0 \text{ else} \end{cases}$$

Typically, γ is one of the following criteria:

- $C_{\max} = \max_{1 \leq i \leq n}\{C_i\}$ is the finish time of the last job.
- $L_{\max} = \max_{1 \leq i \leq n}\{L_i\}$ the maximum lateness.
- $\sum_{i=1}^{n} C_j$ the total flow time
- $\sum_{i=1}^{n} w_i U_i$ the total weight of late jobs

A *schedule* is an allocation of a time interval $[\sigma_{i,j}, \phi_{i,j}]$ on machine $\mu_{i,j}$ to each operation $o_{i,j}$ of all jobs $J_i \in J$. A schedule is *feasible* if no job is processed before its release date (if given), the interval allocated to an operation does not fall short of its specified processing time, no two time intervals allocated to the same job overlap and no two time intervals on the same machine overlap. In addition, a number of specific

requirements concerning machine environment and job characteristic have to be met. In addition, a schedule is *optimal*, if it minimizes a given optimality criterion.

In the sequel, we are mainly interested in job-shop scheduling with *blocking*. Interestingly, job-shop scheduling with blocking is a rather unusual combination. For example, Pinedo states that blocking is a phenomenon that occurs only in flow shops [8] and in the survey article of Hall and Sriskandarajah complexity results only for job shop with no-wait but not with blocking can be found [7]. One reason why most of the research has focused on flow shops may be that most practical applications of *blocking* and *no wait* are in flow shops. However, our traffic control problem is best considered a job-shop problem with blocking. Solutions for such problems have to satisfy the following conditions:

$$\sigma_{i,j} \geq r_i \tag{6}$$

$$\phi_{i,j} - \sigma_{i,j} \geq p_{i,j} \tag{7}$$

$$\phi_{i,j} = \sigma_{i,j+1} \tag{8}$$

$$\mu_{i,j} = \mu_{r,s} \Rightarrow o_{i,j} = o_{r,s} \vee [\sigma_{i,j}, \phi_{i,j}] \cap [\sigma_{r,s}, \phi_{r,s}] = \emptyset \tag{9}$$

Condition (6) states that job i should start at or after the release date of job i and condition (7) requires that the time on machine of the jth subtask of job i is not less than the minimum time required for that subtask. Condition (8) formalizes the *blocking* constraint and, finally, condition (9) states that machines can only be exclusively used.

While these conditions seem to be enough to guarantee that the schedule can be executed (and in fact, for flow-shop problems these conditions are sufficient), in a job-shop environment it might be the case that two jobs with opposite machine routing meet face to face, which is obviously a deadlock and might result in a complete breakdown of the whole system. Therefore, condition

$$\mu_{i,j} = \mu_{r,s+1} \wedge \mu_{i,j+1} = \mu_{r,s} \Rightarrow \phi_{i,j} \neq \sigma_{r,s+1} \tag{10}$$

should also be satisfied. Interestingly, this condition has not been discussed in the scheduling literature yet. The main reason is probably that, as mentioned above, blocking usually happens in flow-shop contexts and the blocking constraint has not been seriously considered for job-shop environments.

To model the traffic problem as a scheduling problem, we consider infrastructural resources as machines, vehicles as jobs and movement activities as operations. Therefore, we have to choose the job shop machine environment $\alpha = J$, which allows us to equate the sequence of movement activities of a vehicle with a jobs sequence of operations. A necessary job characteristic is $\{\text{block}\} \subseteq \beta$, since if a vehicle v_i wants to move from resource $r_{i,j}$ to $r_{i,j+1}$ but $r_{i,j+1}$ is blocked by another vehicle, v_i has to wait on $r_{i,j}$ until $r_{i,j+1}$ becomes available. Finally, the optimality criterion we choose is $\gamma = C_{\max}$, i.e., the minimization of the maximal completion time. In terms of the classification scheme introduced in section 3, $\{J|\text{block}|C_{\max}\}$ is our type of scheduling problem we are going to use for solving traffic problems.

The transformation of a traffic problem into a scheduling problem is straightforward: If $P_{\text{traff}} = (R, V)$ is a traffic problem with resources $R = \{r_1, \ldots, r_m\}$ and vehicles $V = \{v_1, \ldots, v_n\}$ where each vehicle v_i is associated with a movement plan

$[a_{i,1}, \ldots, a_{i,k_i}]$, then $P_{\text{sched}} = (R, V)$ is the corresponding scheduling problem where R is interpreted as a set of machines and V as a set of tasks. Each movement activity $a_{i,j}$ that has to be performed on resource $\rho_{i,j}$ corresponds to an operation $o_{i,j}$ that has to be performed on machine $\mu_{i,j} \in R$. Obviously, a feasible schedule S directly corresponds to a safe traffic flow F. It is obvious that conditions (1) - (5) for safe traffic flows are equivalent to conditions (6) - (10) for feasible schedules.

Proposition 1. *If S is a feasible schedule for a job shop scheduling problem with blocking, then S represents a safe traffic flow for the corresponding traffic problem.*

From this correspondence we can immediately derive a complexity result.

Theorem 1. *The traffic control problem is strongly*[3] *NP-hard if we want to optimize the maximum completion time.*

Proof. As shown by Hall and Sriskandarajah [7], the problem $F_3|\text{blocking}|C_{\max}$ is strongly NP-hard. This however, is clearly a special case of $J_3|\text{blocking}|C_{\max}$, which implies that the traffic control problem with three resources is already strongly NP-hard.

While this result is not surprising, it nevertheless shows that the traffic control problem is a computationally difficult problem. Moreover, the result implies that we should look for heuristic approaches in order to solve it.

4 Solution Existence and the Infrastructure

If we know that regardless of the movements of our vehicles the goals can be reached, we can concentrate on finding a schedule that minimizes the overall costs. Conversely, if it is possible that a system state can be reached from which some vehicles cannot proceed to the goal positions, then we better focus on avoiding such states and consider optimization as secondary.

Let us first consider the situation in Figure 3. Clearly, regardless of what we do, the

Fig. 3. A traffic problem instance without a solution

depicted problem instance does not have a solution. Conversely, if we consider problem instances such that the start and final points are not on the routes of other vehicles, then the problem instance has a solution. The reason is that we could move each vehicle to

[3] Strong NP-hardness means that even if the numbers in the problem description are coded in unary way, the problem remains to be NP-hard.

its final destination, starting with a new vehicle once the starting time has come and the previous vehicle has reached its final destination. This guarantees that we can move all vehicles collision-free to its goals – provided it is enough to start a vehicle movement at some point after its start time. If we have to begin the vehicle movement exactly at the start time, we may run into problems. While the entire restriction sounds very severe, the restriction is satisfied, for example, in *open* infrastructures, such as airports and train stations. For example, at airports we might delay the landing of an airplane for as long as the taxi ways are blocked.

However, even if the problem instance is solvable, it might be possible that a system state is reachable from which the goal cannot be achieved. For example, in Figure 4 a

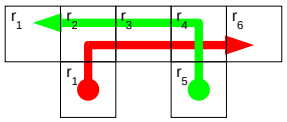


Fig. 4. A traffic problem instance with a possible deadlock situation

situation with a possible deadlock situation is depicted. Such deadlocks can, of course, be anticipated and avoided in the scheduling procedure. However, due to the on-line nature of the problem, it can happen that while executing the activity plan, one vehicle is delayed and the deadlock happens accidentally. In order to avoid that, often the use of resources is restricted in certain ways. For instance, often roads can only be used in a uni-directional way resulting sometimes in detours but avoiding head-on conflicts such as the possible one in Figure 4.

5 Very Fast Approximations for Fast-Time Simulations

In this section, we introduce a very fast algorithm for creating a safe traffic flow for a given traffic problem. Our simulation results indicate that the efficiency of the automatic generated traffic flow can keep up with the efficiency of a traffic flow obtained by a human controller. As a possible application we show how such an automatic traffic controller can be used for assessing the capacity limit of a given infrastructure.

For the corresponding job shop problem with blocking, we built a feasible schedule by incrementally inserting jobs in a first-come-first-served manner into the schedule. The order in which jobs are inserted is determined by their release date. If S is a schedule, then $S_{\mu_j} = \{o_{r,s} \in S | \mu_{r,s} = \mu_j\}$ is the *machine schedule* of $\mu_j \in M$ and $\text{idle}(\mu_j, \sigma, \phi)$ is true if and only if no operation is processed on μ_j during time $[\sigma, \phi]$. Furthermore, $\Phi_{\mu_j} = \{\phi_{r,s} | o_{r,s} \in S_{\mu_j}\}$ is the set of finish times at μ_j and

$$\sigma^*_{i,j} = \begin{cases} r_i & j = 1 \\ \phi_{i,j-1} & j > 1 \end{cases}$$

is the *earliest possible start time* of $o_{i,j}$. If $o_{i,j}$ has to be inserted into schedule $S_{\mu_{i,j}}$, we consider only a finite number of potential start times:

$$\Sigma_{i,j} = \{\sigma^*_{i,j}\} \cup \{\phi \in \Phi_{\mu_{i,j}} | \phi > \sigma^*_{i,j}\}$$

For the example shown in Figure 5, $\Sigma_{i,j} = \{7, 9, 14, 19\}$. Finally, the predicate insertable$(S, o_{i,j}, \sigma)$

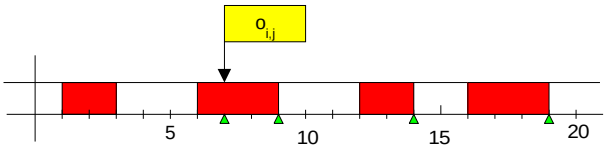


Fig. 5. Potential start times considered for insertion

is true, if and only if in the given schedule S

1. idle$(\mu_{i,j}, \sigma, \sigma + p_{i,j})$ is true, i.e., no other operation is planned on $\mu_{i,j}$ during $[\sigma, \sigma + p_{i,j}]$.
2. Condition (8) can be satisfied, i.e., if $j > 1$, idle$(\mu_{i,j-1}, \phi_{i,j-1}, \sigma)$ has to be true.
3. Condition (10) is not violated, i.e., $o_{i,j}$ does not exchange machines with any other $o_{r,s}$.

As stated above, the basic idea is to sequentially insert jobs into the schedule, so the main algorithm *AutoController* is very simple:

Algorithm AutoController
Params: sequence $(j_1, \ldots, j_n)$ of jobs with $j_i = (o_{i,1}, \ldots, o_{i,k_i})$
Returns: feasible schedule S

$S \leftarrow \emptyset$
for all $j_i \in \{j_1, \ldots, j_n\}$ **do**
  inserted = $false$
  ScheduleActivity$(S, o_{i,1}$, inserted)
end for
return S

For every job j_i, the recursive procedure *ScheduleActivity* is called. In a nutshell, this procedure inserts a given operation $o_{i,j}$ into S and continues recursively with the subsequent operation $o_{i,j+1}$, until finally the last operation o_{i,k_i} has been inserted, causing the boolean variable *inserted* to be set to $true$:

Procedure ScheduleActivity
Params: schedule S, operation $o_{i,j}$, bool *inserted*

if $j \leq k_i$ **then**
  $\Sigma_{i,j} \leftarrow \{\phi^*_{i,j}\} \cup \{\phi \in \Phi_{\mu_{i,j}} | \phi > \sigma^*_{i,j}\}$ *// compute potential start times for* $o_{i,j}$

```
  while Σ_{i,j} ≠ ∅ ∧ ¬inserted do
    σ ← min Σ_{i,j} // get next pot. start time
    Σ_{i,j} ← Σ_{i,j} \ {σ}
    if Insertable(S, o_{i,j}, σ) then
      σ_{i,j} ← σ; φ_{i,j} ← σ_{i,j} + τ_{i,j} // assign start/end time to o_{i,j}
      if j > 1 then
        φ_{i,j-1} ← σ_{i,j} // adapt end time of preceeding operation
      end if
      ScheduleActivity(S, o_{i,j+1}, inserted) // continue recursion with o_{i,j+1}
      if ¬inserted then
        S ← S \ {o_{i,j}}
        if j > 1 then
          φ_{i,j-1} ← σ_{i,j-1} + τ_{i,j-1} // reset end time of preceeding operation
        end if
      end if
    end if
  end while
else
  inserted ← true // last operation of task j_i has been inserted
end if
```

Proposition 2. *For a given job shop scheduling problem with blocking, the* AutoController *algorithm returns a feasible schedule.*

That a solution is returned follows from the fact that a new job can always be inserted at the end of a partial schedule. In particular, for every operation $o_{i,j}$ the set of useful start times $\Sigma_{i,j}$ is never empty and the predicate insertable $(S, o_{i,j}, \max(\Sigma_{i,j}))$ is always true. It is easy to show that conditions (6) - (10) are satisfied after each insertion of an operation, so it follows that the returned schedule is feasible.

6 Experimental Results

We tested the AutoController in a traffic simulation based on the infrastructure partially displayed in Figure 6. It is an open traffic system where vehicles dynamically arrive at entry resources, receive a fixed route randomly taken from a standard route library (with average route length 60 resources) and move along that route to a loading point. After a 10-15 minute stay, they move to an exit resource and leave the traffic system. The minimum time needed to travel along a resource is 10 secs for all vehicles.

Both a human controller and the *AutoController* have been confronted with the same random sequence of 58 vehicles whose start times are equally distributed over 1 hour. The resulting traffic flow contains 123 conflicts and is even for skilled controllers very demanding. Since the latest leave time strongly depends on the arrival times of the last few vehicles, both the human controller and *AutoController* achieved the same completion time. Hence, we use average delay as our secondary optimization criterion that plays an important role for an economic utilization of the traffic system. As can

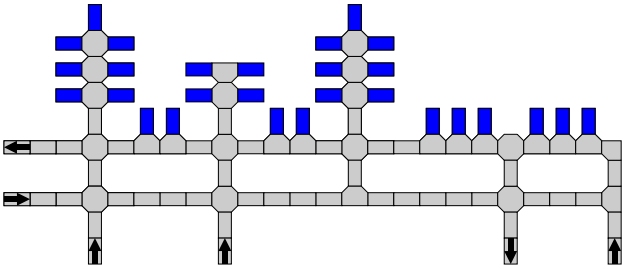
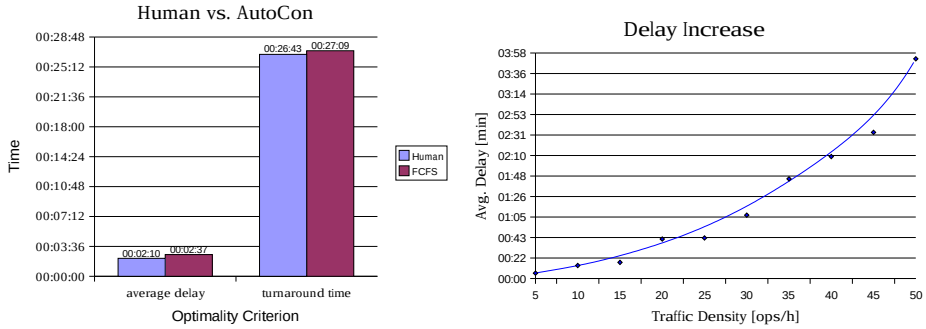


Fig. 6. Simulated infrastructure

be seen in Figure 7 (a), the AutoControllers traffic flow includes 27 secs more average delay than the human controllers traffic flow, which is a difference in performance of 20.77%.



(a) Comparison of human and automatic control (b) Traffic Density and Average Delay

Fig. 7. Simulation Results

However, when taking into account the total time a vehicle spends on the infrastructure (turnaround time), this amounts to a loss of efficiency of merely 1.62%.

Although the *AutoController* is a backtracking algorithm, in our simulation hardly any backtracking occurred. A total of 3429 movement activities has been scheduled and only 83 times an insertion was reversed. Consequently, the algorithm is extremely fast and it took less than 0.5 secs to compute a safe traffic flow on a 300 MHz PC, while on the other hand even a skilled human controller has to run the simulation most of the time in real time which takes about 30-45 minutes.

A typical application that requires the coordination of vehicle movements are fast-time traffic simulations that, among others, are used to evaluate the impact of infrastructural or operational changes in terms of capacity increase or decrease. The capacity limit of a given infrastructure is assessed by gradually increasing traffic density [ops/h] until an acceptable delay limit is exceeded. With the *AutoController* algorithm, the re-

lationship between traffic density and average delay can be determined within minutes, even if an entire day has to be simulated. An example for our infrastructure is shown in Figure 7. In our experiment with the infrastructure as depicted in Figure 6, the capacity limit is 45 [ops/h] if 2:30 min is the acceptable delay limit.

7 Conclusions

The traffic problem is a very common problem occurring when multiple vehicles have to be coordinated. Examples are airport ground traffic coordination, train station coordination, and multi-robot path coordination. We have shown that this problem is a particular kind of scheduling problem, namely, a *job-shop* scheduling problem with *blocking*. This is a rather unusual scheduling problem and it turns out that it is necessary to consider new constraints on schedules, which have not been discussed in the scheduling literature yet, in order to guarantee executability. Nevertheless, the reformulation of the traffic control problem as a scheduling problem allows us to derive the computational complexity of the traffic control problem. Furthermore, on the practical side, the reformulation suggests methods to generate schedules.

We consider restrictions on the problem which guarantee the existence of a solution and we specify a simple, albeit powerful method that is able to generate schedules that are reasonably good. In particular, this method is so fast that it can be used in fast-time traffic simulations, which are needed when doing infrastructure assessments. In an experiment we demonstrate that the simulation method is reasonably good and fast enough to simulate a traffic flow in an infrastructure in a fraction of the time necessary to execute this flow in real-time.

References

1. K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.
2. R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling.* Addison-Wesley, 1967.
3. M. Helmert. On the complexity of planning in transportation and manipulation domains,. Diplomarbeit, Albert-Ludwigs-Universität, Freiburg, Germany, 2001.
4. J. E. Hopcroft, J. T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects: PSPACE-hardness for the 'warehousman's problem'. *Int. J. Robotics Research*, 3:76–88, 1984.
5. J.-C. Latombe. *Robot Motion Planning.* Kluwer, Dordrecht, Holland, 1991.
6. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: algorithms and complexity. In S. C. Graves, P. H. Zipkin, and A. H. G. Rinnooy Kan, eds., *Logistics of Production and Inventory: Handbooks in Operations Research and Management Science*, vol. 4, pp. 45–522. North-Holland, 1993.
7. C. Sriskandarajah N.G. Hall. A survey of machine scheduling problems with blocking and no-wait in process. *Operations Research*, 44(3), 1996.
8. M. Pinedo. *Scheduling - Theory, Algorithms, and Systems.* Prentice-Hall, 1995.
9. J. Lenstra R. Graham, E. Lawler and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.
10. D. Ratner and M. Warmuth. Finding a shortest solution for the (N x N)-extension of the 15-puzzle is intractable. *J. Symbolic Computation*, 10:111–137, 1990.

Toward an Understanding of Local Search Cost in Job-Shop Scheduling

Jean-Paul Watson[1], J. Christopher Beck[2],
Adele E. Howe[1], and L. Darrell Whitley[1]

[1] Colorado State University, Fort Collins, CO 80523-1873 USA
{watsonj,howe,whitley}@cs.colostate.edu
[2] ILOG, S.A., 9, rue de Verdun, B.P. 85
F-94523 Gentilly Cedex France
cbeck@ilog.fr

Abstract. Local search algorithms are among the most effective approaches for solving the JSP, yet we have little understanding of which problem features influence search cost in these algorithms. We study a descriptive cost model of local search in the job-shop scheduling problem (JSP), borrowing from the MAX-SAT cost models. We show that several factors known to influence the difficulty of local search in MAX-SAT directly carry over to the general JSP, including the number of optimal solutions, backbone size, the distance between initial solutions and the nearest optimal solution, and an analog of backbone robustness. However, these same factors only weakly influence local search cost in JSPs with workflow, which possess structured constraints. While the factors for the MAX-SAT cost models provide an accurate description of local search cost in the general JSP, our results for workflow JSPs raise concerns regarding the applicability of cost models derived using random problems to those exhibiting specific structure.

1 Introduction

Local search algorithms, particularly those based on tabu search, are among the most effective approaches for solving the JSP [BDP96]. Yet, we have little understanding as to *why* these algorithms work so well, and under what conditions. In this paper, we study descriptive cost models of local search in the JSP. Descriptive cost models relate search space features to search cost; better models account for more of the variance in search cost across different problem instances. We examine the cost of tabu search in the JSP by considering an algorithm introduced by Taillard (1994), which is closely related to many state-of-the-art algorithms for the JSP (e.g., [NS96]), and is significantly more amenable to analysis.

Although no descriptive cost models for the JSP exist, researchers have expended significant effort in recent years to produce relatively accurate descriptive cost models of local search for MAX-SAT [SGS00]. Intuitively, we would expect some factors present in these models, such as the number of optimal solutions, to influence the difficulty of local search in other problems such as the JSP. At the same time, both the search space and constraint structure of the JSP differ in many important ways from MAX-SAT, making the a-priori applicability of these models unclear.

We investigate whether or not the descriptive cost models for MAX-SAT can be leveraged in an effort to understand local search cost in the JSP. We demonstrate that the factors present in the MAX-SAT cost models also influence local search cost in the JSP, including the number of solutions [CFG+96], backbone size [Par97], the distance between initial solutions and the nearest optimal solution [SGS00], and an analog of backbone robustness [SGS00]. Together, these factors form the basis of a relatively accurate descriptive model of local search cost in the *general* JSP.

The constraints in both MAX-SAT (clauses) and the general JSP (machine processing orders) are randomly generated, and in expectation are unstructured. In contrast, the constraints in real-world problems are often structured. We apply the same analysis to JSPs with workflow, a restricted form of the JSP with simple, structured constraints. We find that the factors present in the MAX-SAT and general JSP descriptive cost models only weakly influence search cost in workflow JSPs. We conclude by discussing the implications of our analysis for the JSP, MAX-SAT, and local search in general.

2 The JSP and Problem Difficulty

We consider the well-known $n \times m$ static JSP, in which n jobs must be processed exactly once on each of m machines for an arbitrary, pre-specified duration. Each machine can process only one job at a time, and once initiated, processing cannot be interrupted. Any job can start at time 0, and the objective is to minimize the *makespan*, or the maximum completion time of any job. In the *general* JSP, the machine processing orders are independently sampled from a uniform distribution. In the *workflow* JSP, machines are typically divided into two equal-sized partitions containing machines 1 through $m/2$ and $m/2 + 1$ through m, respectively, and every job must be processed on all machines in the first partition before any machine in the second partition. Within each partition, the machine processing orders are sampled from a uniform distribution.

While no descriptive cost models for the JSP exist, some general qualitative observations regarding problem difficulty have emerged. First, independent of local search algorithm, we have the following trends:

1. For both general and workflow JSPs, "square" ($n/m \approx 1$) problem instances are significantly harder than "rectangular" ($n/m \gg 1$) problem instances.
2. Given fixed n and m, workflow JSPs are substantially more difficult than general JSPs.

Second, given either general or workflow JSPs with a fixed n and m, the relative difficulty of problem instances appears to be algorithm-independent: e.g., a problem instance that is difficult for tabu search is likely to be difficult for simulated annealing. Clearly, any descriptive cost model for the JSP must be consistent with each of these observations.

Mattfeld et al. (1999) perform a quantitative analysis of problem difficulty in the JSP, identifying significant differences in the search spaces of some well-known 50×10 general and workflow JSPs. Specifically, they show that the extension of the search space (as measured by the average distance between random local optima) is larger in workflow JSPs, suggesting a cause for the generally larger search cost associated with these problem instances. Two other measures, entropy and correlation length, also demonstrated quantitative differences in the search spaces of these same problems.

While Mattfeld et al. do identify differences in the search spaces of general and workflow JSPs, it is unclear whether these differences account for the variance in local search cost for different problem instances of the same size and workflow configuration. In preliminary experiments, we found that while the factors introduced by Mattfeld et al. *did* influence search cost, the influence was much weaker than for the factors we discuss in Section 5. Furthermore, Mattfeld et al. did not investigate whether these same factors were responsible for the relative difficulty of square versus rectangular JSPs.

3 The MAX-SAT Descriptive Cost Model

The MAX-SAT descriptive cost model is the basis for our study. Many methods for characterizing problem difficulty have found application in a wide variety of problems, for example phase transitions and the associated peak in search cost. Given such universals, it is important to examine, and if possible leverage, any existing analysis on problem difficulty. However, the literature on local search yields descriptive cost models for only two, related problems: MAX-SAT and MAX-CSP. Outside of these two examples, the dominant methods for quantifying problem difficulty are unable to account for the large cost variance found in different problem instances of a given size. For example, correlation length [RS01] is strictly a function of problem size (e.g., the number of cities in the TSP).

Intuitively, a decrease in the number of optimal solutions should yield an increase in local search cost. This observation formed the basis of the first descriptive cost model for MAX-SAT, in which Clark et al. (1996) demonstrated a relatively strong (negative) log-log correlation between the number of solutions and local search cost, with r-values ranging anywhere from -0.77 to -0.91. However, the model failed to account for the large cost variance in problems with very small numbers of optimal solutions, where model residuals varied over three or more orders of magnitude.

Singer et al. (2000) subsequently introduced a descriptive cost model that largely corrected the deficiency present in the Clark model, and went further by proposing a causal model for local search cost in MAX-SAT. The *backbone* of a problem instance is a key concept in Singer's descriptive cost model. The backbone of a MAX-SAT instance consists of the subset of literals that have the same truth value in *all* optimal solutions[Par97]. Singer demonstrated that the backbone size does influence search cost in MAX-SAT, showing that when the backbone is small, there is a strong (negative) log-log correlation ($r \approx -0.77$) between the number of optimal solutions and the local search cost. However, this correlation nearly vanishes ($r \approx -0.12$) when the backbone is large [SGS00].

Local search algorithms for MAX-SAT quickly locate sub-optimal *quasi-solutions*, that contain relatively few unsatisfied clauses. These quasi-solutions form a sub-space that contains all optimal solutions, and is largely interconnected; once a point in this sub-space is identified, local search algorithms for MAX-SAT typically restrict search to this sub-space. This observation led Singer to hypothesize that the size of this sub-space dictates the overall search cost, which could overcome the inability of the number of optimal solutions to predict local search cost in problems with large backbones.

To test this hypothesis, Singer measured the mean Hamming distance (the number of differing variable assignments) between the first quasi-solution encountered during

local search and the nearest optimal solution, which we denote $d_{init-opt}$, and computed the correlation between $d_{init-opt}$ and the logarithm of local search cost. The resulting correlations were extremely high ($r \approx 0.95$) for problems with small backbones, and degraded only slightly for problems with larger backbones ($r \approx 0.75$). Consequently, $d_{init-opt}$, and not the number of optimal solutions, is the primary factor influencing local search cost in MAX-SAT.

Singer also posited a causal explanation for the variance in $d_{init-opt}$ across different MAX-SAT instances, which is based on the notion of *backbone robustness*. A MAX-SAT instance is said to have a *robust* backbone if a substantial number of clauses can be deleted before the backbone size is reduced by half. Conversely, an instance is said to have a *fragile* backbone if the deletion of just a few clauses reduces the backbone size by half. Singer argues that "backbone fragility approximately corresponds to how extensive the quasi-solution area is" ([SGS00], p. 251), by noting that a fragile backbone allows for large $d_{init-opt}$ because of the sudden drop in backbone size, while $d_{init-opt}$ is necessarily small in problem instances with robust backbones.

To confirm this hypothesis, Singer measured a moderate (≈ -0.5) negative correlation between backbone robustness and the log of local search cost for large-backboned MAX-SAT instances. Surprisingly, this correlation degraded as the backbone size was decreased, leading Singer to hypothesize that "finding the backbone is less of an issue and so backbone fragility, which hinders this, has less of an effect" ([SGS00], p. 254), although this conjecture was not explicitly tested.

4 Algorithms, Test Problems, and Methodology

We now briefly describe Taillard's tabu search algorithm for the JSP, and introduce the test problems and methodology we use to investigate descriptive cost models for the JSP.

4.1 Algorithm Description

The tabu search algorithm we consider in our analysis was introduced by Taillard (1994). This was the first tabu search algorithm for the JSP and is the basis for more advanced, state-of-the-art JSP algorithms such as that of Nowicki and Smutnicki (1996). Taillard's algorithm uses the Van Laarhoven move operator [LAL92], which is often denoted by *N1*. The *N1* neighborhood is generated by swapping all adjacent pairs of jobs on any critical path in the current solution. As in most tabu search algorithms for the JSP, recently swapped pairs of jobs are prevented from being re-established for a particular duration, called the tabu tenure; the tabu tenure is dynamically updated to avoid cycling behavior. All runs are initiated from randomly generated "active" solutions [GT60]. In each *iteration* of Taillard's algorithm, all *N1* neighbors are generated, and the best non-tabu move is taken. The only long-term memory mechanism is a simple aspiration criterion, which over-rides the tabu status of any move that results in a solution that is better than any encountered in the current run. As Taillard indicates ([EDT94], p. 110), frequency-based long-term memory is only necessary for problems that require a very large (> 1M) number of iterations, which is not the case for the test problems introduced later in this section.

The cost required to solve a given problem instance using Taillard's algorithm is naturally defined as the number of iterations required to locate an optimal solution. However, the number of iterations is stochastic (with an approximately exponential distribution [EDT94]), due to both the randomly generated initial solution and random tie-breaking when more than one 'best' move is available. Consequently, we define the local search cost for a problem instance as the median number of iterations required to locate an optimal solution over 1000 independent runs; with 1000 samples, the estimate of the distribution median is somewhat stable [Hoo98] [SGS00].

For analysis purposes, the most important feature of Taillard's algorithm is the *N1* move operator. More advanced critical path move operators for the JSP, such as that used in Nowicki and Smutnicki's algorithm, can induce search spaces that are *disconnected*, such that it is not always possible to move between a randomly generated solution and an optimal solution. Consequently, any algorithm using such a move operator is not Probabilistically Approximately Complete (PAC) [Hoo98]: even with infinite run-time, the algorithm is not guaranteed to locate an optimal solution. This severely complicates algorithm analysis, as it is unclear how to define the search cost associated with a problem instance. However, use of a connected move operator does *not* automatically guarantee that an algorithm is PAC [Hoo98]. While we have no analytic proof that Taillard's algorithm is PAC, the empirical evidence is compelling: in producing the results discussed in Sections 5 and 6, Taillard's algorithm never failed to locate an optimal solution.

4.2 Defining a Backbone for JSP

The definition of a backbone in any problem depends on how the solutions are represented. Taillard's algorithm encodes solutions using a *disjunctive graph*, which contains $n(n-1)/2$ Boolean "order" variables for each of the m machines, each of which represents a precedence relation between a distinct pair of jobs on a machine. We define the *backbone* of a JSP, therefore, as the set of order variables that have the same truth value in all optimal solutions. We define the *backbone size* as the fraction of the possible $mn(n-1)/2$ order variables that are fixed to the same value in all optimal solutions.

4.3 Test Problems

For a variety of reasons, we are restricted to relatively small problem sizes in our experiments. From a technical standpoint, the factors present in our descriptive cost model (e.g., backbone size and the distance between initial and optimal solutions) are functions of *all* optimal solutions to a problem instance. Generating all optimal solutions to a problem instance is much more expensive than merely proving optimality (the enumeration is explicit, in contrast to the implicit approach characteristic of branch-and-bound algorithms). Further, the number of optimal solutions in even small problem instances is measured in the millions, which can easily exceed available memory in modern workstations. From a pragmatic standpoint, we consider a wide range of problems in our experiments, controlling for backbone size, workflow configuration, and problem size. We study over 4000 problem instances, computing the median search cost over 1000 independent runs for each. Even for the problem sizes we consider, the overall CPU time invested was approximately 8 CPU months on 750 MHz Pentium III workstations.

In our experiments, we examine 6×4 and 6×6 problems, both with and without workflow partitions. Operation durations are sampled uniformly from the interval $[1, 99]$. Backbone size is an integral factor in our descriptive cost model. Unfortunately, even for these problem sizes, it is infeasible to control for a *specific* backbone size: computation of the backbone is considerably more expensive in the JSP than in MAX-SAT. Instead, we filter for problems within $\pm 5\%$ of a target backbone size X, $0.0 \leq X \leq 1.0$. We denote the backbone size of the resulting set of problems by $\approx X$. For each problem size, we generated 100 general and workflow JSP instances at each of the following backbone sizes: ≈ 0.1, ≈ 0.3, ≈ 0.5, ≈ 0.7, and ≈ 0.9. Finally, we used a constraint-directed scheduling algorithm to compute the optimal makespan, the backbone size, and to enumerate all optimal solutions. The specific algorithm is documented in Beck and Fox (2000), which uses min-slack variable and value ordering heuristics and edge-finding constraint propagators.

5 A Descriptive Cost Model for the General JSP

A-priori, it is unclear whether the factors present in the MAX-SAT cost models are relevant to local search in the JSP. In MAX-SAT, the search space is dominated by plateaus of equally-fit quasi-solutions, and the main challenge for local search is to either find an exit from a plateau to an improving quasi-solution, or to escape the plateau by accepting a short sequence of dis-improving moves [FCS97]. In contrast, the JSP search space is dominated by local optima with variable-sized and variable-depth attractor basins. Consequently, local search algorithms for the JSP spend much of their time either escaping or avoiding local optima. In this section, we examine the MAX-SAT cost models in the context of Taillard's algorithm for the JSP, and demonstrate that despite qualitative differences in search space topologies, the MAX-SAT cost model factors *do* form the basis of an accurate descriptive model of local search in the JSP.

5.1 Number of Solutions and Search Cost

In MAX-SAT, the number of optimal solutions *can* influence local search cost, although the strength of this influence depends critically on backbone size: it is strong in problems with small backbones, and very weak in problems with large backbones. In Table 1, we report summary statistics for the number of optimal solutions and the local search cost for our general JSPs. We see both a dramatic drop in the number of optimal solutions and a gradual increase in local search cost as the backbone size is increased. Further, at a fixed backbone size the difference in local search cost between the 6×6 and 6×4 problems is minimal, and can be attributed to the larger size of the search space in the 6×6 problem instances.

In the bottom third of Table 1, we report the log_{10}-log_{10} correlation between the number of optimal solutions and local search cost. The r-values indicate that both the number of optimal solutions and the backbone size influence local search cost in the general JSP. As in MAX-SAT, the correlation is relatively strong for small-backboned problems, and drops rapidly with increases in backbone size. Although additional factors are required to fully account for the variance in local search cost for large-backboned general JSPs, these results demonstrate that the interaction effect between backbone size and the number of solutions is not unique to MAX-SAT.

Problem Size	Backbone Size				
	≈ 0.1	≈ 0.3	≈ 0.5	≈ 0.7	≈ 0.9
	Number of Optimal Solutions				
6×4	481837 ± 1158660	30007 ± 38072	3221 ± 3742	642 ± 1374	21 ± 22
6×6	6233821 ± 8070114	1405290 ± 3221150	85292 ± 157617	9037 ± 9037	85 ± 106
	Local Search Cost				
6×4	6.69 ± 3.34	23.05 ± 20.93	52.64 ± 61.22	94.76 ± 136.56	312.27 ± 332.27
6×6	8.34 ± 4.13	32.94 ± 32.58	53.43 ± 58.52	83.98 ± 91.80	514.70 ± 1853.08
	log_{10}-log_{10} Correlation (r) Between the # of Optimal Solutions and Local Search Cost				
6×4	-0.7508	-0.5100	-0.4905	-0.4131	-0.2683
6×6	-0.7328	-0.4865	-0.3807	-0.3227	-0.2010

Table 1. The number of optimal solutions, local search cost, and log_{10}-log_{10} correlation (r) between the number of optimal solutions and local search cost for general JSPs. $X \pm Y$ denotes a mean of X with a std. dev. of Y.

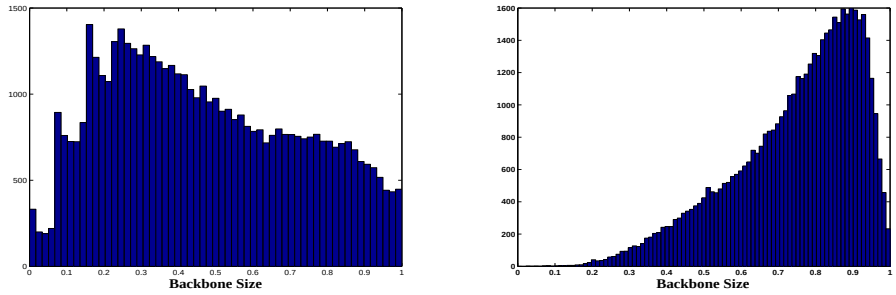


Fig. 1. Histogram of backbone sizes for 50 000 6×4 (left figure) and 6×6 (right figure) general JSPs.

5.2 Distribution of Backbone Sizes

While rectangular JSPs tend to be much easier than square JSPs, this difference was not observed in the local search costs reported in Table 1. In a straightforward experiment, we generated 100 6×4 and 6×6 general JSPs and computed the local search cost for each problem set, leaving the backbone size *uncontrolled*. The mean local search costs were 32.91 and 498.13 for the 6×4 and 6×6 problem sets, respectively, suggesting a strong bias in the distribution of backbone sizes for the two problem types.

In MAX-SAT, the distribution of backbone sizes depends on the ratio of the number of clauses c to the number of variables v [Par97]. Under-constrained problem (with small values of c/v) tend to have small backbones, while over-constrained problems (with large values of c/v) tend to have large backbones; the relative frequency of large-backboned problems increases rapidly in the so-called 'critically constrained' region. In the JSP and many other optimization problems, there is no known parameter analogous to c/v by which we can control for the expected degree of constrainedness. Consequently, we can only observe the relative frequency of backbone sizes in these problems.

To examine the relative frequency of backbone sizes in the general JSP, we generated 50 000 6×4 and 6×6 problems, and computed the backbone size for each instance. In Figure 1, we provide histograms illustrating the relative frequency of the backbone sizes. The most common backbone sizes for the square 6×6 instances are roughly 0.9, and are exceedingly rare below 0.3. In contrast, the backbone sizes for the rectangular 6×4 instances are more uniformly distributed, with a slight bias toward smaller backbone

Problem	Backbone Size				
Size	≈ 0.1	≈ 0.3	≈ 0.5	≈ 0.7	≈ 0.9
$d_{init-opt}$-log_{10}(local search cost) correlation					
6×4	0.9890	0.9526	0.9070	0.8296	0.5303
6×6	0.9912	0.9327	0.8911	0.8371	0.6484
Backbone robustness-log_{10}(local search cost) correlation					
6×4	-0.2193	-0.3993	-0.4412	-0.5277	-0.5606
6×6	-0.1621	-0.3629	-0.4507	-0.4712	-0.5134

Table 2. Correlation (r) of 1) $d_{init-opt}$ and 2) backbone robustness with log_{10}(local search cost) in general JSPs.

sizes. We have also generated similar histograms for other small problem sizes: for ratios of $n/m > 1.5$, the bias toward small backbones becomes more pronounced, while for ratios < 1, the bias toward larger backbones is further magnified. Finally, we note that the utility of the correlation between number of optimal solutions and local search cost depends heavily on problem size; the influence is negligible for nearly all 6×6 JSPs (which generally have large backbones), and for many 6×4 JSPs.

5.3 Distance to Global Optima and Search Cost

In MAX-SAT, the mean distance between the initial quasi-solutions encountered by local search and the nearest optimal solution ($d_{init-opt}$) is strongly correlated with local search cost, across all backbone sizes. Intuitively, we would also expect the distance between the first local optima encountered by local search and the nearest optimal solution to influence local search cost in the JSP; the question is then "How strong is this influence?".

For each of our general JSPs, we generated 1000 local optima, computed the Hamming distance to the nearest optimal solution for each of the resulting optima, and recorded the mean of the 1000 distances (the Hamming distance between two solutions in the JSP is the number of order variables, out of the $mn(n-1)/2$ possible, with different assigned values); as with MAX-SAT, we denote this measure by $d_{init-opt}$. We generated the local optima by applying a next-descent algorithm from random "active" solutions [GT60]. Our next-descent algorithm evaluates the neighbors of the current solution under the *N1* move operator in a random order, selecting the first solution that improves on the makespan of the current solution; the algorithm terminates when no such improvements are possible.

In Table 2, we report the correlations between $d_{init-opt}$ and log_{10}(local search cost). For backbone sizes of ≈ 0.1 through ≈ 0.5, the correlation is extremely high, and only moderately degrades for the two larger backbone sizes. The r-values are uniformly and significantly better than those achieved using the number of solutions, and account for a significant proportion of the variance in local search cost for large-backboned problems. Thus, we also conclude that the distance between initial and optimal solutions, and not the number of optimal solutions, is the primary factor influencing the cost of local search in the general JSP, independent of backbone size.

5.4 Backbone Robustness and Search Cost

Singer et al. propose backbone robustness a causal factor that largely determines the size of the quasi-solution sub-space in MAX-SAT. Abstractly, backbone robustness is a

measure of the number of problem constraints that must be relaxed to produce a problem with a significantly smaller backbone. While in the JSP there is no analog to relaxing individual constraints (as is possible in MAX-SAT), there is a parameter controlling the global constrainedness: deviation from the optimal makespan. Thus, we define backbone robustness for the JSP as the minimum percentage above the optimal makespan at which the backbone size is reduced by at least half (subject to integral makespan constraints).

In the lower half of Table 2 we report the correlation between the backbone robustness and log_{10}(local search cost) for our general JSPs. The results are very similar to those reported by Singer et al. for MAX-SAT; a moderate negative correlation for large-backboned instances, and a gradual decay as backbone size is decreased. Analogous to MAX-SAT, backbone robustness does appear to partially dictate the size of the subspace containing local optima in the general JSP. As we noted in Section 3, Singer et al. provide a justification for the lower correlations for small-backboned instances.

6 Extending the Analysis to Workflow JSPs

The primary problem constraints in the JSP are the machine processing orders for each job, while in MAX-SAT they are the individual clauses. In both cases, researchers typically generate problem instances such that these constraints are uniformly random. An important issue is then generalization: real-world problems have non-random constraints, and it is unclear whether the descriptive cost models for MAX-SAT and general JSP are applicable to problem instances with more structured constraints. To study the effect of non-random constraints on the accuracy of the descriptive cost model, we extend the analysis of Section 5 to JSPs with workflow–which impose a simple, specific structure on the machine processing orders for each job.

First, we consider the influence of the number of optimal solutions on local search cost in workflow JSPs, reported in Table 3. As with general JSPs, we see both a dramatic drop in the number of optimal solutions and a gradual increase in local search cost as the backbone size is increased. Workflow JSPs have significantly fewer optimal solutions than general JSPs, and the local search cost is generally an order of magnitude higher. However, the log_{10}-log_{10} correlation between the number of optimal solutions and the local search cost is nearly identical with the results for general JSPs: correlation is strong for small-backboned problems, but decays as backbone size is increased.

Next, we computed the relative frequency of backbone sizes for both the 6×4 and 6×6 workflow JSPs; the resulting histograms are shown in Figure 2. Relative to general JSPs (Figure 1), it is clear that the presence of workflow partitions dramatically increases the frequency of large-backboned problem instances. For the rectangular 6×4 problems, workflow changes a bias toward small backbones in the general JSP into a relatively large bias toward large backbones. For the 6×6 problems, workflow further magnifies the already large bias toward large backbones found in the general JSP. We note that the rarity of small-backboned workflow JSPs further diminishes the utility of the number of solutions as a predictor of local search cost for these instances.

Finally, we measured the correlation between $d_{init-opt}$ and log_{10}(local search cost); the results are reported in the upper portion of Table 4. Here, we see a dramatic difference between general JSPs and workflow JSPs: while the influence of $d_{init-opt}$ at

Problem Size	Backbone Size				
	≈ 0.1	≈ 0.3	≈ 0.5	≈ 0.7	≈ 0.9
	Number of Optimal Solutions				
6 × 4wf	27369 ± 71049	81255 ± 295593	2515.25 ± 4704	293 ± 425	18 ± 16
6 × 6wf	1147650 ± 6555440	429102 ± 1676350	19017 ± 44556	4553 ± 6898	80 ± 94
	Local Search Cost				
6 × 4wf	119.44 ± 89.32	122.2 ± 114.26	333.42 ± 442.39	920.72 ± 1515.75	2087.44 ± 2973.86
6 × 6wf	318.77 ± 113.82	513.13 ± 143.72	1086.33 ± 1979.39	1730.53 ± 2846.15	5036.53 ± 5132.54
	log_{10}-log_{10} Correlation (r) Between the # of Optimal Solutions and Local Search Cost				
6 × 4wf	-0.7650	-0.6663	-0.3484	-0.2613	-0.2208
6 × 6wf	-0.7345	-0.6877	-0.4316	-0.2700	-0.2561

Table 3. The number of solutions,local search cost, and log_{10}-log_{10} correlation (r) between the number of solutions and local search cost for JSPs with workflow. $X \pm Y$ denotes a mean of X with a std. dev. of Y.

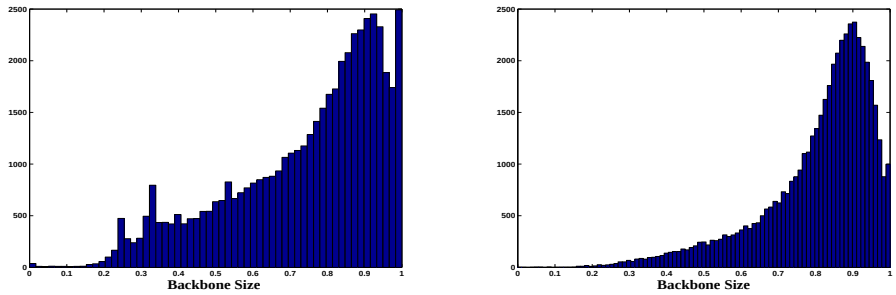


Fig. 2. Histogram of backbone sizes for 50 000 6 × 4 (left figure) and 6 × 6 (right figure) JSPs with workflow.

small backbones is relatively large, it drops very rapidly, ultimately vanishing at ≈ 0.9. Additionally, because of the lesser influence of $d_{init-opt}$ on local search cost, we see a corresponding drop in the influence of backbone robustness, as shown in the bottom half of Table 4. Given the strong bias toward large backbones in workflow JSPs, we conclude by noting that the factors present in the MAX-SAT and general JSP cost models are unable to account for *any* significant proportion of the variance in local search cost in these problems.

7 Discussion and Implications

Our results demonstrate that $d_{init-opt}$ is a good predictor of local search cost in both the general JSP and MAX-SAT, despite qualitative differences in the underlying search spaces. In both cases, $d_{init-opt}$ indirectly measures the size of the search space explored by the respective local search algorithms. Modern local search algorithms for MAX-SAT (e.g., Walk-SAT [SGS00]) basically perform a random walk over the quasi-solution sub-space. Consequently, it is unsurprising that search cost is an exponential function of the sub-space size [Hoo98]. However, this inference also applies to Taillard's tabu search algorithm for the general JSP: it is effectively performing a random walk in the space of local optima. The ability of $d_{init-opt}$ to predict local search cost also indicates that there is no, or at most a very weak, bias in the search spaces of both problems; if there were, distance alone would fail to accurately predict local search cost.

Problem Size	Backbone Size				
	≈ 0.1	≈ 0.3	≈ 0.5	≈ 0.7	≈ 0.9
$d_{init-opt}$-log_{10}(local search cost) correlation					
6 × 4wf	0.8727	0.7122	0.5109	0.1811	0.0862
6 × 6wf	0.8231	0.6781	0.5264	0.1367	0.0711
Backbone robustness-log_{10}(local search cost) correlation					
6 × 4wf	-0.0029	-0.0217	-0.0372	-0.0752	-0.1423
6 × 6wf	-0.0165	-0.0348	-0.0513	-0.0941	-0.1239

Table 4. Correlation (r) of 1) $d_{init-opt}$ and 2) backbone robustness with log_{10}(local search cost) in workflow JSPs.

In contrast, we found that $d_{init-opt}$ was a very poor predictor of local search cost in workflow JSPs. Follow-up experiments indicate that there is a very strong bias toward particular sub-optimal solutions in some of these problems: there are many more 'paths' in the search space to sub-optimal solutions than to optimal solutions. In other problems, we have observed very distant clusters of optimal solutions, suggesting that a more complicated definition of $d_{init-opt}$ may be required. Our results also raise issues regarding the descriptive cost models for MAX-SAT, as we have shown that the factors influencing local search cost in random and structured problems *may* in fact be quite different.

We view this research as a first step toward understanding why local search algorithms for the JSP are so effective. We selected Taillard's tabu search algorithm precisely because it serves as a baseline for more advanced algorithms, such as Nowicki and Smutnicki's tabu search algorithm, which enhance Taillard's algorithm through either more advanced move operators or long-term memory. With descriptive cost models for the basic algorithm, we can begin to *systematically* assess the influence of these improvements on the descriptive cost model. Finally, we note that our analysis is only directly applicable to tabu-like search algorithms for the JSP. Because descriptive cost models are tied to specific algorithms, it seems likely that other factors are responsible for local search cost in algorithms such as iterated local search or genetic algorithms, which are based on principles quite different from tabu search.

8 Conclusions

Our results clearly demonstrate that the factors influencing local search cost in MAX-SAT also influence local search cost in the general JSP, despite qualitative differences in the underlying search spaces. Consequently, we have a relatively clear picture of local search cost in the general JSP, although our model fails to account for a moderate amount of the variance in local search cost of large-backboned problem instances. Our results also suggest the possibility that these same factors may be applicable in a much wider range of optimization problems.

We also shed more light on the observation that rectangular JSPs are significantly easier than square JSPs. If we control for backbone size, rectangular JSPs are *not* significantly easier than square JSPs. Instead, the observed difference in difficulty stems primarily from the relative frequency of backbone sizes in the two problems: large backbones are very common in square problems, while we see a bias toward smaller backbones in rectangular problems.

Finally, we also demonstrate that the factors influencing search cost in the general JSP do not necessarily transfer to JSPs with workflow, suggesting that the descriptive cost models for random and structured problems may in fact be quite different.

Acknowledgments

The authors from Colorado State University were sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-00-1-0144. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. J. Christopher Beck would also like to thank Paul Shaw (ILOG, S.A.) for discussions relating to this work.

References

[BDP96] Jacek Blażewicz, Wolfgang Domschke, and Erwin Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93:1–33, 1996.

[BF00] J. Christopher Beck and Mark S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(2):31–81, 2000.

[CFG+96] David A. Clark, Jeremy Frank, Ian P. Gent, Ewan MacIntyre, Neven Tomov, and Toby Walsh. Local search and the number of solutions. In *Proceedings of the Second International Conference on Principles and Practices of Constraint Programming (CP-96)*, pages 119–133, 1996.

[EDT94] Éric D. Taillard. Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing*, 6(2):108–117, 1994.

[FCS97] Jeremy Frank, Peter Cheeseman, and John Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

[GT60] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8(4):487–503, 1960.

[Hoo98] Holger H. Hoos. *Stochastic Local Search - Methods, Models, Applications.* PhD thesis, Darmstadt University of Technology, 1998.

[LAL92] P.J.M Van Laarhoven, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40:113–125, 1992.

[MBK99] Dirk C. Mattfeld, Christian Bierwirth, and Herbert Kopfer. A search space analysis of the job shop scheduling problem. *Annals of Operations Research*, 86:441–453, 1999.

[NS96] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[Par97] Andrew J. Parkes. Clustering at the phase transition. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 340–345, 1997.

[RS01] Christian M. Riedys and Peter F. Stadler. Combinatorial landscapes. Technical Report 01-03-014, The Santa Fe Institute, 2001.

[SGS00] Josh Singer, Ian P. Gent, and Alan Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.

Flexible Integration of Planning and Information Gathering

David Camacho, Daniel Borrajo, José M. Molina, and Ricardo Aler

Universidad Carlos III de Madrid, Computer Science Department, Avenida de la Universidad nº 30, CP 28911, Leganés, Madrid, Spain
{dcamacho, dborrajo, molina}@ia.uc3m.es, aler@inf.uc3m.es

Abstract. The evolution of the electronic sources connected through wide area networks like Internet has encouraged the development of new information gathering techniques that go beyond traditional information retrieval and WEB search methods. They use advanced techniques, like planning or constraint programming, to integrate and reason about hetereogeneous information sources. In this paper we describe MAPWEB, a multiagent framework that integrates planning agents and WEB information retrieval agents. The goal of this framework is to deal with problems that require planning with information to be gathered from the WEB. MAPWEB decouples planning from information gathering, by splitting a planning problem into two parts: solving an abstract problem and validating and completing the abstract solutions by means of information gathering. This decoupling allows also to address an important aspect of information gathering: the WEB is a dynamic medium and more and more companies make their information available in the WEB everyday. The MAPWEB framework can be adapted quickly to these changes by just modifying an abstract planning domain and adding the required information gathering agents. For instance, in a travel assistant domain, if taxi companies begin to offer WEB information, it would only be necessary to add new planning operators related to traveling by taxi, for a more complete travel domain. This paper describes the MAPWEB planning process, focusing on the aforementioned flexibility aspect.

1 Introduction

In recent years there has been a lot of work in Web information gathering [1, 5–8]. Information gathering intends to integrate a set of different information sources with the aim of querying them as if they were a single information source [6]. Many different kinds of systems, named *mediators*, have been developed. They try to integrate information from multiple distributed and heterogeneous information sources, like database systems, knowledge bases, web servers, electronic repositories... (an example is the *SIMS* [7] architecture). In order that these systems are practical, they must be able to optimize the query process by selecting the most appropriate WEB sources and ordering the queries. For this purpose, different algorithms and paradigms have been developed. For instance, *Planning by Rewriting* (*PbR*) [1] builds queries by using planning techniques.

Other examples of information gathering systems are Ariadne [7], Heracles [8], WebPlan [5].

Some of the previous approaches use planning techniques to select the appropriate WEB sources and order the queries to answer generic user queries. That is, they use planning as a tool for selecting and sequencing the queries. In this paper we describe MAPWEB, an information gathering system that also uses planning, but with a different purpose (some preliminary work can be found in [2, 3]). MAPWEB uses planning for both determining the appropriate generic sources to query and solving actual planning problems. For instance, in this paper, the MAPWEB framework is applied to a travel planning assistant domain (e-tourism),[1] where the user needs to find a plan to travel among several places. Each plan not only determines what steps the user must perform, but which information sources should be accessed. For instance, if a step is to go from A to B by plane, the system provides the user the information of what airplane companies should be consulted for further information. This domain is similar to the travel planning assistant built using the Heracles framework. However, Heracles constrained network, which is a kind of plan schema, needs to be reprogrammed everytime the planning domain changes. MAPWEB tries to be more flexible by using planning techniques to create the plans. For instance, if it is desired to add a new information source to the system, it is only necessary to change the planning domain instead of reprogramming the plan schema by hand. For instance, if taxi fares were made suddenly available in the WEB, it would only be necessary to add a move-by-taxi operator along with the associated WebAgent.[2] Actually, MAPWEB can handle planning operators which are not associated to any information source (because, for instance, the information on a given topic is not yet available). In that case, plans will contain steps with no detailed information. This is useful, because even if no specific information is supplied, at least the user is told that he can fulfill that step by any means.

The paper is structured as follows. Section 2 describes MAPWEB architecture. Section 3 explains in detail the abstract planning process. Section 4 evaluates empirically the system. Finally, Section 5 summarizes the conclusions and future lines of work.

2 MAPWEB System Architecture

MAPWEB is structured into several logic layers whose purpose is to isolate the user from the details of problem solving and WEB access. More specifically, we considered four layers between users and the WEB: *the physical world* (the users), *the reasoning layer* (that includes user agents, planning agents, and control agents), *the access information layer* (that contains WebAgents to retrieve the desired information), and *the information world* (which represents the available information). This four-layer architecture can be seen in Figure 1.

[1] This domain is a modified version of the Logistics domain.

[2] A WebAgent is an information agent specialized in consulting a particular information source.

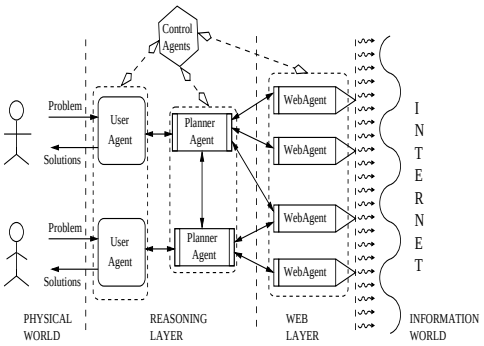


Fig. 1. MAPWeb three-layer architecture.

MAPWeb deploys this architecture using a set of heterogeneous agents. Next, each of these types of agents will be described:

- **UserAgents**: They pay attention to user queries and display to users the solution(s) found by the system. When an UserAgent receives problem queries from the users, it sends them to the PlannerAgents and when they answer back with the plans, the UserAgent provides the solutions to the user.
- **ControlAgents**: They handle several control functions like the insertion and deletion of agents in the system and communication management.
- **PlannerAgents**: They receive an user query, build an abstract representation of it, and solve it by means of planning. Then, the PlannerAgents fill in the information details by querying the WebAgents. The planner that has been used by the PlannerAgents is PRODIGY4.0 [9].
- **WebAgents**: Their main goal is to fill in the details of the abstract plans obtained by the PlannerAgents. They obtain that information from the Web.

The way these agents cooperate is as follows. First, the user interacts with the UserAgent to input his/her query. The query captures information like the departure and return dates and cities, one way or return trip, maximum number of transfers, and some preference criteria. This information is sent to the PlannerAgent, which transforms it into a planning problem. This planning problem retains only those parts that are essential for the planning process, which is named the *abstract representation* of the user query. PRODIGY4.0 generates several abstract solutions for the user query. The planning operators in the abstract solutions require to be completed and validated with actual information which is retrieved from the Web. To accomplish this, the PlannerAgent sends information queries to specialized WebAgents, that return several records for every information query. Then, the PlannerAgent integrates and validates the solutions and returns the data to the UserAgent, which in turn displays it to the user. MAPWeb agents use a subset of the KQML speech acts [4]. The whole process will be described in full detail in the next section.

3 The Planning Process

As mentioned before, in MAPWeb, the information gathering process is carried out by a set of WebAgents, but this process is guided by the PlannerAgent that reasons about the requested problem and the different information sources that are available.

The planning process is divided into two parts: solving an abstract problem, and completing it with information gathered from the Web. Planning is decoupled this way because of two reasons:

- The abstract planning problem is easier to solve by classical planners. This is because if all the information about all the available flights, all possible trains, etc. was included in the planning process, planning would be unfeasible.
- It is not necessary to access the Web during the planning process. Queries to the WebAgents are carried out only when abstract plans are ready. This allows to reduce the number of queries, because only those queries that are required by the solution are ever made.

Planning works as follows. First, the PlannerAgent receives a query from UserAgent. This query is analyzed and translated into an abstract planning problem. Second, the PlannerAgent uses its own skills and knowledge about the problem and tries to solve it. If the solving process is successful, the PlannerAgent generates a set of abstract solutions. These solutions are too general and only have the essential information for the planning process, so they need specific information to be completed and validated. The PlannerAgent builds a set of information queries (queries to other agents in the system to request specific information). It is important to try to optimize the number of queries due to the high number of possible instantiations. When the queries have been built, the PlannerAgent selects the set of WebAgents that will be asked. Finally, when the WebAgents answer with the information found in the Web (if WebAgents are successful) the PlannerAgent integrates all the specific information with the abstract solutions to generate the final solutions that will be sent to the UserAgent. In Figure 2 the modular description of the planning process is shown.

The next subsections explain this process in detail by focusing in the data structures used by each of the relevant agents: the user query generated by the UserAgent, the abstract problem, the abstract solutions, the specific knowledge used by the PlannerAgent, and finally the specific information records retrieved by the WebAgents.

3.1 The User Query

The planning process starts when the user supplies a problem to be solved. A user query is a sequence of stages. Each stage is a template that represents a leg of the trip, and contains several fields to be filled by the user. Table 1 shows an instance of a possible user query. It will be used to illustrate the rest of the

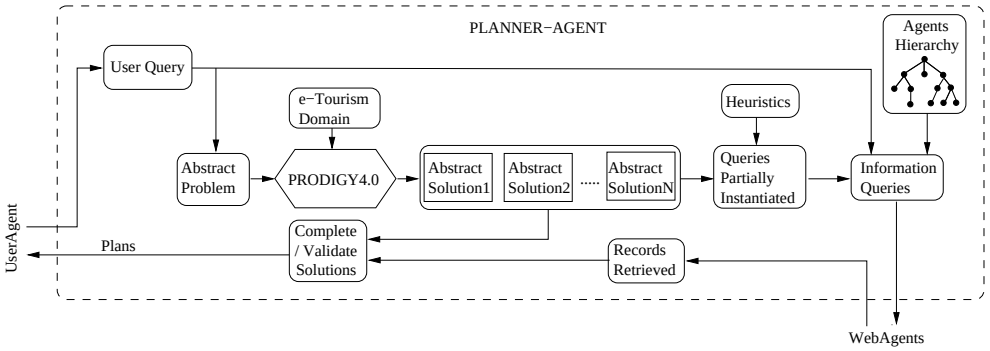


Fig. 2. Planning Process developed by the PlannerAgent. The user query is transformed into an abstract planning problem, which is subsequently solved by PRODIGY4.0. Each solution is partially instantiated by means of domain dependent heuristics. Every operator in a solution generates several WEB queries, which are sent to the appropriate WebAgents by using the agent hierarchy. The agents return several records, that are used to complete and validate the abstract solutions.

article. This query is then sent to the PlannerAgent. Besides the information shown in Table 1, the user can specify the locations inside the city where s/he wants to start or end the trip (like an airport, a train station, or a bus station). This is done by means of the user interface provided by the UserAgent.

Table 1. A user problem to go from Turin to Toledo by airplane or train.

Leg	Stage	Date	Restrictions	N° Transfers
1	Turin → Madrid	Sep. 11th	Plane or train	0 or 1
2	3 nights stay	Sep. 11th	< 15.000 pts	-
3	Madrid → Toledo	Sep. 14th	Plane or train	0 or 1
4	Toledo → Turin	Sep. 14th	Plane or train	0 or 1

3.2 The planning domain and the abstract solutions

The PlannerAgent transforms the user query into an abstract problem. This is done as follows. First, it defines an abstract city. This city includes all possible local transports, but only the long range transport terminals that the user wishes to use are included. Then, this abstract city is copied as many times as the maximum number of transfers supplied by the user. It is important to remark that the cities are abstract cities (i.e. they have no attached names, so they are present in the abstract plan to represent the initial, intermediate, and final travel points). The rest of details provided by the user are ignored at this stage. The abstract problem represents the initial state and the goals of the problem that are the inputs to PRODIGY4.0.

In order to solve abstract problems, PRODIGY4.0 requires a domain where the planning operators are described. Using planning at this stage (instead of using pre-programmed plans) provides two main advantages:

1. *Flexibility:* the system can be adapted to many different versions of travel domains and problems by just changing the domain description or the abstract problem generation method, respectively.
2. *Easy integration of new* WEB *sources.* The WEB is a dynamic medium: more and more companies make their information available in the WEB everyday. If a new information source (like taxi fares) is made available, MAPWEB can be adapted quickly by just adding a new planning operator and establishing a relation with a WebAgent specialized in gathering the information from the WEB.

The abstract problem obtained from Table 1 would be given to the PlannerAgent planner (PRODIGY4.0) which would obtain several possible abstract solutions. In this case, the planner would reply with the plans shown in Figures 3 (solutions with 0 transfers) and 4 (1 transfer solutions).

```
Problem: 0-Transfers
Solution 1:
 <travel-by-airplane user1 plane0 airport0 airport2>
 <move-by-local-transport user1 lbus2 bustop20 trainstat21 city2>

Solution 2:
 <move-by-local-transport user1 lbus0 bustop00 trainstat01 city0>
 <travel-by-train user1 train0 trainstat0 trainstat2>
```

Fig. 3. Abstract solutions generated by PRODIGY4.0 for Leg 1 with 0-Transfer.

```
Problem: 1-Transfers
Solution 1:
 <travel-by-airplane user1 plane0 airport0 airport1>
 <move-by-local-transport lbus1 bustop10 trainstat11 city1>
 <travel-by-train user1 train1 trainstat1 trainstat2>

Solution 2:
 <travel-by-airplane user1 plane0 airport0 airport1>
 <travel-by-airplane user1 plane1 airport1 airport2>
 <move-by-local-transport lbus2 bustop20 trainstat20 city2>
..............
```

Fig. 4. Abstract solutions generated by PRODIGY4.0 for Leg 1 with 1-Transfers.

This is a set of abstract plans that contain no actual details. Some of the plan steps might not even be possible because, for instance, there are no companies linking two cities. Therefore, those plans need to be *validated* and *completed.* The PlannerAgent acomplishes this task in the following way:

1. The abstract steps in the solution contain unbound variables that relate to transfer cities. They need to be bound before the WebAgents are queried.

The PlannerAgent restricts the number of bindings by applying a *geographic heuristic*. This is achieved as follows:

- If the origin and arrival cities belong to the same country, only the cities in that country are considered as possible transfer cities.
- Else, if the origin and arrival cities belong to the same continent, only the cities of that continent are considered.
- Otherwise, all cities are considered.

In the case of the first leg of the trip, as Turin and Madrid belong to Europe, we extract the cities that belong to this continent (currently, about 30). Table 2 displays the queries that would be generated in this case.

Table 2. Queries partially instantiated.

Query send to the WebAgents	N° Transfers
(travel-by-airplane user1 plane0? Turin Toledo)	0
(travel-by-train user1 train0? Turin Toledo)	0
(travel-by-airplane user1 plane0? Madrid Turin)	0
(travel-by-train user1 train0? Madrid Toledo)	0
(travel-by-airplane user1 plane0? Turin Alicante)	1
(travel-by-airplane user1 plane0? Turin Barcelona)	1
(travel-by-airplane user1 plane0? Turin Paris)	1
(travel-by-train user1 train0? Turin Madrid)	1
...	...

2. Planning operators of the abstract solutions and WEB sources are related by means of a WebAgent *hierarchy*. This *hierarchy* is used by the PlannerAgent to select the relevant WebAgents that will be used to obtain the information. This hierarchy allows the PlannerAgent to know which WebAgents know how to retrieve the required information. In Figure 5 a description of this hierarchy is shown.

```
Agent
    |___UserAgent0
    |---ReasonerAgent
                  |___PlannerAgent0
    |___WebAgents
              |___Travel
                        |___Fly: WebAgent-Iberia,WebAgent-Amadeus-Flight
                        |___Train: WebAgent-Renfe,WebAgent-RailEurope
                        |___Bus
              |___Hotel: WebAgent-Amadeus-Hotel
              |___Car
    |___ControlAgents
                  |___ManagerAgent0
                  |___CoachAgent0
```

Fig. 5. Agents Hierarchy. It describes all the available agents in MAPWEB and their information gathering skills.

3. Finally the PlannerAgent uses the previous information to build a set of queries that will sent to the selected WebAgents. If a planning operator is repeated in different abstract solutions, it is only considered once, to avoid repeating queries. For instance, in the solutions for 1-Transfer problems the operator (<`travel-by-airplane user1 plane0 airport0 airport1`>) would be translated as shown in Table 3:

Table 3. Queries partially instantiated to the appropriate WebAgents.

Query send to the WebAgents	WebAgent
(travel-by-airplane user1 plane0? Turin Toledo)	Iberia, Amadeus-Flights
(travel-by-train user1 train0? Turin Toledo)	Renfe, RailEurope
(travel-by-airplane user1 plane0? Turin Alicante)	Iberia, Amadeus-Flights
(travel-by-airplane user1 plane0? Turin Barcelona)	Iberia, Amadeus-Flights
(travel-by-airplane user1 plane0? Turin Paris)	Iberia, Amadeus-Flights
(travel-by-train user1 train0? Turin Madrid)	Renfe, RailEurope
...	...

Those queries (and all the additional information given by the UserAgent) are sent to several WebAgents that know about airplane travel, so that variable `plane0?` is instantiated as well.

3.3 Filling the Abstract Solutions

The information queries are sent to the selected WebAgents with the specific data (departure and arrival times, travel cost, etc...) and the query that the PlannerAgent needs. With this information the WebAgents automatically build the specific WEB query that will be sent to the WEB information sources the agent is specialized in. For every query, each WebAgent will return to the PlannerAgent a list of *records* by filling a template whose structure is shared by all the agents (there are different templates depending on the kind of information required). In Table 4 some of the retrieved flight-records and train-records provided by different WebAgents are shown for the Leg 1 in the example.

Finally, those records are received by the PlannerAgent that will use them to complete the abstract solutions. If the WebAgents return no records for a step of the abstract solution, that particular solution is rejected. However, it is important to remark that if it is known in advance that there are no WEB sources to complete a particular step (for instance, <`MOVE-BY-LOCAL-TRANSPORT taxi ...`>), then the user is told that s/he has to carry out that step, even though no specific information about that step is attached. The set of completed solutions are finally sent to the UserAgent that requested the information.

4 Experimental Evaluation

The aim of this section is to carry out several experiments with MAPWEB to evaluate its performance. First, the example-trip we have used to illustrate the

Table 4. Retrieved records by the WebAgents.

Inf-FLIGHTS	record1	record2	record3	Inf-TRAINS	record1	record2	record3
WebAgent	**Iberia**	**Amadeus**	**Amadeus**	**WebAgent**	**Renfe**	**Renfe**	**Renfe**
air-company	Iberia	Iberia	Portugalia	train-company	RENFE	RENFE	RENFE
http-address	w3.iberia.es	null	null	http-address	w3.renfe.es	w3.renfe.es	w3.renfe.es
flight-id	IB8797	IB8819	NI711	train-id	07054	07056	07058
ticket-fare	70641	null	null	ticket-fare	780	780	780
currency	ESP	ESP	ESP	currency	ESP	ESP	ESP
flight-duration	3h45min	2h00min	2h10min	departure-city	MAD	MAD	MAD
airp-depart-city	TRN	TRN	TRN	departure-date	11-09-01	11-09-01	11-09-01
departure-date	11-09-01	11-09-01	11-09-01	departure-time	6:30	8:30	10:14
airp-arrival-city	MAD	MAD	MAD	arrival-city	TOL	TOL	TOL
return-date	null	null	null	arrival-date	11-09-01	11-09-01	11-09-01
class	Tourist	null	null	arrival-time	7:53	9:47	11:30
n° passengers	1	1	1	class	Tourist	Tourist	Tourist
round-trip	one-way	one-way	one-way				

previous sections will be tested. Second, a set of problems given by the user will be evaluated to analyze the average behaviour of the system.

Table 5 summarizes the example-trip (Turin to Toledo and back). To solve this problem, a team of nine agents was used. It includes all the agents displayed in the agents-hierarchy of Figure 5. In particular, airplane, train, and hotel WebAgents have been used. The next parameters have been measured:

- Validated abstract solutions/abstract solutions ratio (*val.sols/abs.sols*). This value measures how many abstract solutions provided through the planner were validated by the information provided by the information gathering agents.
- Number of instantiated solutions. It shows all the possible solutions to the user problem. The solutions are computed using the gathered records. The PlannerAgent uses the k validated abstract solutions that contain l_i abstract operators. If there are b_{ij} retrieved records for the j-th operator of the i-th solution, then the number of possible instantiated solutions is:

$$\text{Number of solutions} = \sum_{i=1}^{k} \prod_{j=1}^{l_i} b_{ij}$$

- Number of WEB Queries. This represent all the queries made by the WebAgents to retrieve the specific information.
- Number of gathered records (duplicated records are removed).
- Time. It includes planning time and WEB gathering time. It is elapsed time (i.e. the time spent by the WebAgents acting in parallel is not accumulated).

Everyone of the previous parameters is measured for both 0 and 1 transfers (0-T and 1-T). In this example, there are no solutions for the 0 transfers because it is impossible to complete the fourth leg of the trip (there is no way to go from Toledo to Turin directly). On the other hand, there are thousands of possible combinations when 1 transfer is allowed. It is important to remark that even though when 1 transfer is used, it takes several thousand seconds to find the solutions, only 1 second per leg is spent for actual planning.

Table 5. MAPWeb request for the example-trip, with 0 and 1 transfers.

Leg	Stage	Val. sols per abs. sols		Number of solutions		Number of queries		Number of records		Time (seconds)	
		0-T	1-T	0-T	1-T	0-T	1-T	0-T	1-T	0-T	1-T
1	Turin → Madrid	0.5	0.667	2	1829	4	43	20	135	112.465	962.938
2	3 nights stay	1	1	6	6	1	1	20	20	62.384	62.384
3	Madrid → Toledo	0.5	0.333	12	797	4	43	22	92	75.030	1692.839
4	Toledo → Turin	0	0.333	0	432	4	43	0	67	76.236	3874.948

We have also tested a set of 38 problems with different configurations of MAPWeb. The problems include 15 trips within Spain, 15 within Europe, and 8 Intercontinental ones. Each problem has been tried with 0 and 1 transfers. The results are shown in Table 6. This experiment shows in practice the flexibility of MAPWeb when it is necessary to add new information sources. The configurations that have been used are as follows:

- *N0*: only one WebAgent specialized in retrieving information from a particular Airplane Company (Iberia Airlines[3]) was considered.
- *N1*: different WebAgents specialized in gathering information of the same kind (flight information) were used: WebAgent-Iberia, WebAgent-Avianca, WebAgent-Amadeus-Flights, WebAgent-4Airlines-Flights. The two last ones are meta-searchers.
- *N2*: only two WebAgents specialized in gathering information of the same type (train information) were used: WebAgent-Renfe, WebAgent-RailEurope.
- *N3*: integrates all the previous WebAgents, that is, agents for retrieving both flight and train information (*N3=N1+N2*).

Table 6. Summary of the results for 38 user problems, with 0 and 1 transfers (0-T and 1-T).

Config.	Number of solutions		Solved problems		Number of queries		Time (seconds)	
	0-T	1-T	0-T	1-T	0-T	1-T	0-T	1-T
N0	7.1	999.3	65.7%	74.3%	1	26.9	65.6	1485.7
	σ = 6.7	1480.5					σ = 10.2	1319.2
N1	10.9	1338.1	94.2%	97.1%	4	91.2	162.4	2243.6
	σ = 8.0	1725.8					σ = 200.5	1321.8
N2	3.7	3.7	25.7%	40.0%	2	51.4	70.1	1314.4
	σ = 7.9	7.9					σ = 23.0	1124.3
N3	12.5	1340.2	94.3%	97.1%	6	143.9	165.3	2666.6
	σ = 8.8	1724.4					σ = 199.6	1298.8

In Table 6, we observe the following:

- With respect to *N0*, as it could be expected, many more solutions are found when 1 transfer legs are allowed (999.3 vs. 7.1). It can also be observed that

[3] www.iberia.com/iberia_es/home.jsp

MAPWeb cannot find a solution for some problems, although the number of problems solved increases for the 1-T option (74.3% vs. 65.7%). However, the number of queries and the time required to fulfill them also increases quickly. It is also noticeable that standard deviations are rather large. This is because user problems can be very different; some of them can be solved quickly because there are few retrieved records, whereas other problems can have many possible solutions.

- *N1* enlarges *N0* by including more airplane companies. MAPWeb does not find many more solutions per problem, because most of the user problems are within Europe, where Iberia (the only agent in *N0*) offers many flights. However, many more problems are solved (94.2% vs. 65.7% with 0 transfers, and 97.1% vs. 65.7% for 1 transfer). Although the number of queries is multiplied by 4 in *N1*, the time required to fulfill them has been only doubled (162.4 vs. 65.6 for 0-T and 2243.6 vs. 1485.7 for 1-T). Time is doubled because even though the four WebAgents work in parallel, all the retrieved records must be analyzed by a single PlannerAgent.
- *N2* displays the results when only train travels are allowed. Only a few problems can be solved: 25.7% with 0-T and 40.0% with 1-T, and very few solutions per problem are found (3.7). This is clearly due to the smaller number of possibilities of fullfilling travels using only trains vs. using airplanes.
- *N3* integrates both airplane and train companies. Compared to *N1*, almost the same number of user problems are solved (94.3% vs. 94.2% and 97.1% vs. 97.1%), although some more solutions per problem are found (12.5 vs. 10.9 and 1340.2 vs. 1338.1).

5 Conclusions

The Web is a dynamic medium: more and more companies make their information available in the web everyday. Web information gathering systems need to be flexible to adapt to these rapid changes. In this paper we have described MAPWeb, a multiagent framework that combines classical planning techniques and Web information retrieval agents. MAPWeb decouples planning from information gathering, by splitting a planning problem into two parts: solving an abstract problem and validating and completing the abstract solutions by means of information gathering. Flexible information gathering is achieved by means of planning. In order to add a new information source to the system, only the planning domain has to be modified, besides adding the related Web agent.

In this paper MAPWeb has been applied to the e-tourism domain, but we believe it could be also used in other domains where planning can be separated from Web information gathering. For instance, currently many companies are thinking on moving to the Web and most organization process models will be implemented in such a way that they use information stored in the Web (either information internal to the organization or external). These processes can be automatically generated on-the-fly by planners, and they will need the information stored in the Web to decide on the steps to be performed. For

instance, one might define what information to publish (and how) in the WEB depending on the competence prices. This publishing process could be generated automatically by a planner.

In the future, several new skills will be developed for different agents in MAPWEB. These skills will try to improve the performance of the global system in two ways: by increasing the number and quality of solutions found by the agents, and by minimizing the time and computational resources used by MAPWEB to solve problems.

Acknowledgements

The research reported here was carried out as part of the research project funded by CICYT TAP-99-0535-C02.

References

1. Ambite, J.L., Knoblock, C.A.: Planning by rewriting: Efficiently generating high-quality plans. In proceedings of the Fourteenth National Conference on Artificial Intelligence (1997).
2. Camacho, D., Molina, J.M., Borrajo, D.: A Multiagent Approach for Electronic Travel Planning. Proceedings of the Second International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2000). AAAI. July (2000). Austin, TX (USA).
3. Camacho, D., Molina, J.M., Borrajo, D., Aler, R.: MAPWEB: Cooperation between Planning Agents and Web Agents. Information&Security: An International Journal. Special issue on Multi-agent Technologies. Volume 7 (2001).
4. Finin, T., Fritzson, R., Mackay, D., McEntire, R.: KQML as an Agent Communication Language. In Proceedings of the Third International Conference on Information and Knowledge Management (CIKM94), pages 456–463. New York: Association of Computing Machinery (1994).
5. Hüllen, J., Bergmann, R., Weberskirch, F: WebPlan - Dynamic planning for domain-specific search in the Internet. In J. Köhler (Hrsg.) 13. Workshop "Planen und Konfigurieren". (PuK-99) (1999).
6. Lambrecht, E., Kambhampati, S.: Planning for Information Gathering: A tutorial Survey. ASU CSE Techincal Report 96-017. May (1997).
7. Knoblock, C.A., Minton, S., Ambite, J.L., Ashish, N.: Modeling Web Sources for Information Integration. Proceedings of the Fifteenth National Conference on Artificial Intelligence, 1998.
8. Knoblock, C.A., Minton, S., Ambite, J.L., Muslea, M., Oh, J., Frank, M.: Mixed-Initiative, Multi-source Information Assistants. The Tenth International World Wide Web Conference (WWW10). ACM Press. May 1-5. (2001).
9. Veloso, M., Carbonell, J., Perez, A. Borrajo, D., Fink, E., Blythe, J.: Integrating planning and learning: The Prodigy architecture. Journal of Experimental and Theoretical AI. Volume 7 (1995) 81–120.

Supply restoration in power distribution systems – a benchmark for planning under uncertainty

Sylvie Thiébaux[1] and Marie-Odile Cordier[2]

[1] Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia
Sylvie.Thiebaux@anu.edu.au
[2] IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
Marie-Odile.Cordier@irisa.fr

Abstract. This paper proposes the problem of supply restoration in faulty power distribution systems as a benchmark for planning under uncertainty. This benchmark, which is derived from a significant real-world case, is both simple to understand and easily scalable. The goal is to reconfigure the distribution network to resupply a maximum of consumers affected by the faults. Due to sensor and actuator uncertainty, the location of the faulty areas and the current network configuration are only partially observable. This makes the problem very challenging.

1 Motivation

The use of poor benchmarks for planning under uncertainty has often been pointed out as detrimental to the impact of the field on the wider community. Except for a few testbeds in robot navigation, see e.g. [6], we are still confined to purely artificial problems ranging from escaping the tiger behind the door to making an omelette. While well-understood toy problems are definitely useful in explaining performance differences, it is commonly acknowledged that the danger of such experimentation alone is that it "entices us into solving problems that we understand rather than ones that are interesting" [9].

It is rather paradoxical that the literature on planning under uncertainty features so few benchmarks derived from significant real world cases. After all, the main point of the research line was to better address the necessities of applications, and indeed a lot of realistic problems are modelled quite naturally as partially observable Markov decision processes. If this state of affairs is paradoxical, it is also excusable: planning under uncertainty and in particular partial observability has so far resisted our attempts at building algorithms that scale up, leaving no alternative but experimentation "in the tiny".

Fortunately, the latest advances in using compact symbolic representations for planning under uncertainty, e.g. [5, 7, 10, 13], hold promise of the situation being about to change. It is likely that the present decade will see fairly generic

planners dealing with problems involving uncertainty on a scale that was far out of reach until now. It is therefore a timely moment to offer concrete challenges to the field by introducing benchmarks that are of practical significance.

This paper describes the problem of restoring supply in a faulty power distribution system, a problem which is of major concern for electricity distributors. It consists in localizing the faulty lines on the distribution network and reconfiguring the network so as to isolate these lines and resupply most consumers. This has to be done within minutes. When reconfiguring, a few parameters such as breakdown costs should ideally be optimised, without violating capacity constraints and overloading parts of the network. More importantly for our purpose, the sensors used to locate the faults and report the current configuration, as well as the actuators used to change configuration, are not always reliable. This leads to missing information about the current state of the network.

In virtue of this accumulation of realistic features, the problem is an ideal testbed for systems claiming to address the necessities of the real-world. One of its advantages compared to other realistic ones is that it is relatively simple to understand. Only a few straightforward classes of components and actions are involved. Further, the topology of power distribution systems makes it easy to scale the problem up or down in order to assess the efficiency of algorithms. However, despite this simplicity, the size of real distribution systems makes them very challenging for methods developed not only in the planning community, but also in related areas, such as model-based diagnosis, repair and reconfiguration.

The paper is organised as follows. Section 2 describes physical characteristics of power distribution systems. Section 3 explains the problem of supply restoration and details the features that makes this problem a challenging and representative testbed. Section 4 gives an overview of the scope of the problem with respect to existing work in the literature, and Section 5 lists the material that will be made available on the benchmark's webpage. Our description of power distribution systems and of the supply restoration problem is based on work done in 1994-1996 in the framework of a contract between IRISA and the French electricity utility Electricité de France (EDF).[1]

2 Power Distribution Systems

2.1 Topology

A power distribution system, as in Figure 1, can be viewed as a network of electric lines connected via switching devices (SDs), represented by small squares in the figure, and fed via circuit-breakers (CBs), represented by large squares. Switching devices and circuit-breakers are connected to at most two lines. They have two possible positions: either open or closed. White devices in the figure are open, see e.g. SD61, and the others are closed, see e.g. SD60. A circuit-breaker supplies power iff it is closed, and a switching device stops the power propagation

[1] We thank our collaborators at EDF, in particular Isabelle Delouis-Jacob, Olivier Jehl and Jean-Paul Krivine.

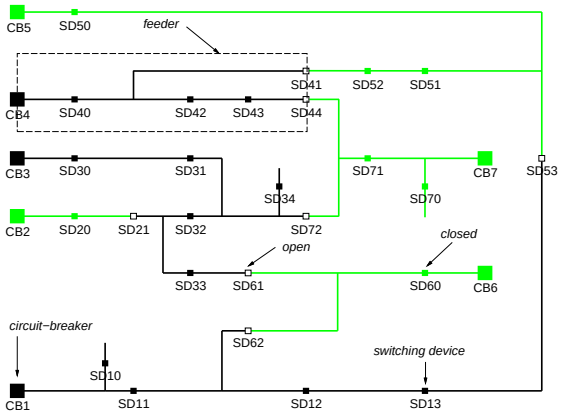


Fig. 1. Power Distribution System (part)

iff it is open. Consumers may be located on any line, and are then only supplied when their line is supplied.

Distribution networks have a meshable structure exploited radially: the positions of the devices are set so that the paths taken by the power of each circuit-breaker form a tree called a feeder. The root of a feeder is a circuit-breaker, and its leaves are whatever switching devices downstream happen to be open at the time. In most cases, each line belongs to one feeder at a time.[2] For illustration, the boxed area in the figure shows one of the feeders, and adjacent feeders are distinguished using alternately black or grey.

2.2 Faults

Power distribution systems are often subject to permanent[3] faults (short circuits) occurring on one or even several lines. Since these short circuits are mainly due to bad weather conditions and lightning, multiple faults are not rare. Upon occurrence of a fault, the circuit-breaker feeding the faulty line opens in order to protect the rest of its feeder from damaging overloads. For instance, if a fault occurs on the line between SD12 and SD13, CB1 will open. As a result, all consumers located on that feeder are left without power. Simply reclosing the circuit-breaker will not help. Since the fault is permanent, the circuit-breaker will still be feeding it and will open again. Instead, using the sensors and actuators described below, the faulty lines must be located and the network reconfigured so as to isolate them and restore the supply to the non-faulty lines. This has to be done within a few minutes.

[2] In certain circumstances, it is possible for a line to be fed by multiple circuit-breakers, i.e., to be belong to more than one feeder. In that case, these circuit-breakers are leaves of each other's feeder.

[3] Technically, a permanent fault is one that cannot be eliminated by automatic protection devices such as shunts and reclosers.

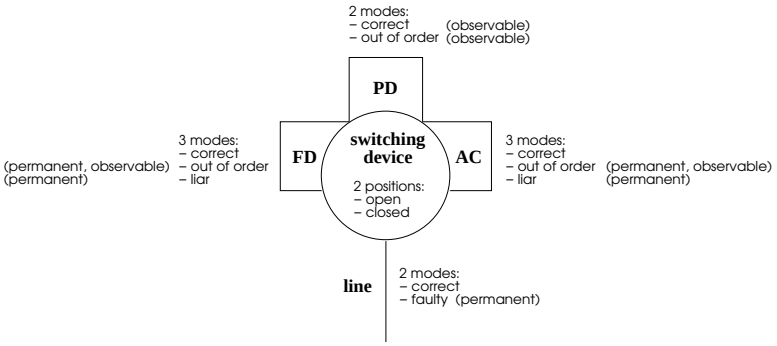


Fig. 2. Possible states and modes of the network components

2.3 Sensors and actuators

As shown in figure 2, switching devices are equipped with a remote-controlled actuator (AC) used to change their position, a position detector (PD) sensing this position, and a fault detector (FD) sensing the presence of faults. Circuit-breakers are only equipped with the former two.

In normal operation, fault detectors work as follows. As long as a switching device is fed, its fault detector constantly indicates whether or not it has "seen a fault pass" i.e., whether or not a fault is downstream of the device on the feeder.[4] If the device is not fed, its fault detector retains the status it had when last fed. For instance, if the line between SD12 and SD13 is faulty, SD11 and SD12 should indicate a fault while the other devices on this feeder should not. Then CB1 should open and the fault detectors' information should remain the same until they are fed again. So, normally, a fault is located on the line between a sequence of switching devices whose fault detectors indicate that it is downstream and a sequence of switching devices whose fault detectors indicate that it is not.

Note that in the case of multiple faults on the same feeder, only the most downstream faults will be detected. A more significant problem is that fault detectors are not always correct and can be in one of the following two permanent abnormal modes. In "out of order" mode, they do not provide any information. Obviously, this mode is observable. In "liar" mode, they always lie, i.e. indicate the negation of the correct reading. That mode cannot be directly observed. Due to these abnormalities, the fault location cannot be identified with certainty on the sole basis of the information returned by the fault detectors.

The primary role of actuators is to open switching devices so as to isolate suspected lines and close others to direct the power from other feeders towards the non-faulty lines. In fact, opening and closing devices are the only available actions in our problem. In normal mode, an actuator executes the requested switching operation and returns a positive notification. Actuators are not always

[4] In the rare event when a switching device belongs to multiple feeders, it indicates a fault if there is one downstream with respect to at least one of the feeders.

reliable and can be in one of the following permanent abnormal modes: "out of order", i.e. the actuator fails to execute the operation and sends a negative notification, or "liar", i.e. it fails to execute the operation but sends a positive notification. The former mode is observable while the latter is generally not.

The continuous information provided by the position detectors often removes uncertainty about the success of switching operations positively notified by actuators. However, position detectors too can be "out of order". In that mode, they do not return any information for an indeterminate time, during which, even though the mode is observable, the network configuration remains uncertain. Figure 2 summarizes the various modes of the network components.

2.4 Size

Like many European power distribution systems, EDF networks are composed of some hundreds of feeders (typically from 100 to 300), each of which contains a few remote-controlled switching devices (the objective is to equip each feeder with 2-3 of them). A feeder has only a very few neighbours (typically from 1 to 4), and as will be seen below, essentially only those will play a role in supply restoration. Hence reasoning is very local, and the network in figure 1 is a good representative of the complexity of the real problem. The problem can trivially be scaled up or down by modifying the number of switching devices per feeder and the number of neighbours of feeders. For experimentation purposes, it should be easy to generate random variations of existing networks.

3 The problem

3.1 Supply restoration

The problem of supply restoration is that of reconfiguring the network in order to resupply the consumers following the loss of one or more feeders. It amounts to building a restoration plan consisting of opening/closing operations. This plan should isolate the faulty lines by prescribing to open the switching devices surrounding them. It should also restore supply to the non-faulty areas of the lost feeders by prescribing to operate devices so as to direct the power towards these areas. Note that although we use the term restoration *plan*, there is no constraint on the nature of the plan (linear, conditional, etc ...) nor a requirement to commit in advance to more than the next action to execute.

The following capacity constraint determines which restoration plans are admissible: at any time, circuit-breakers and lines can only support a certain maximal power. This might prevent directing the power through certain paths and resupplying all the non-faulty areas. In this paper, we will add another constraint which is not present in the original problem but will considerably reduce the search space: we only consider plans which extend existing feeders. That is, the plan should not transfer any of the load that a healthy feeder had at the time of the incident to another feeder. Other types of plans are very rarely used in practice because they require a complex protocol with the dispatching center.

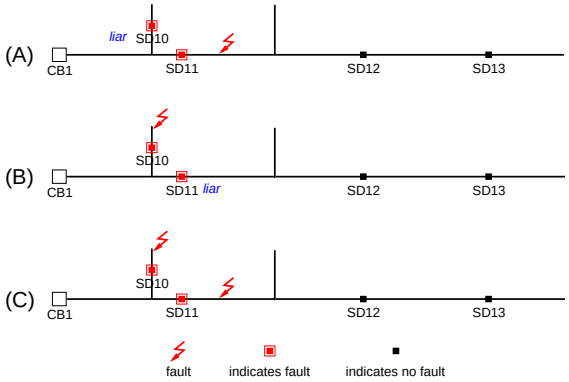


Fig. 3. Three likely hypotheses

A good plan will optimize certain parameters under these constraints. Naturally, breakdown costs should be minimised, i.e., as much load as possible should be resupplied as fast as possible, with priority being given to critical consumers like hospitals. Ideally the number of switching operations should also be optimised, so as to stay close to the configuration in which the network is normally exploited (called the normal configuration), and power margins of circuit-breakers should be balanced in anticipation of the next load peak.

Obviously, the identification of the faulty lines is crucial to the success of the restoration. However, as explained above, this cannot be done on the basis of the information provided by the fault detectors alone. Even in the single fault case, several hypotheses of fault location exist, each of which corresponds to an hypothesis concerning the behaviour mode ("correct", "liar") of the fault detectors. There exist preferences between hypotheses: the probability of multiple faults is much smaller than that of a fault detector lying, and this latter probability is higher when the fault detector indicates a fault downstream than when it does not because fault detectors do not detect all types of faults. But in fact, only reconfiguration actions may enable us to gather enough information to discriminate, as illustrated in the example below.

3.2 Example

Taking our example network, suppose that CB1 opens, leaving the bottom-most feeder on the figure unsupplied. Suppose further that the fault detectors of SD10 and SD11 indicate a fault downstream, while the other fault detectors do not. Among the most probable diagnoses are those shown in Figure 3: (A) the fault detector of SD10 lies and there is only one fault, located between SD11 and SD12, (B) the fault detector of SD11 lies and there is only one fault, downstream of SD10, and (C) none of the fault detectors lie and there are two faults.

Assuming that we consider (A) to be the most likely, a promising restoration plan goes as follows: isolate the line between SD11 and SD12 by opening these devices, resupply the lines upstream of the fault by reclosing CB1, and have CB5 resupply the downstream lines by closing SD53.

Suppose the execution of the plan proceeds correctly up to the point where CB1 reopens when we attempt to close it. Either this is due to a wrong fault location hypothesis (e.g., (B) or (C) was the case), or to a fault which could not be detected (here, a fault between CB1 and SD11), or to a failure of an operation meant to isolate the fault (the actuator of SD11 could be lying, and if its position detector is out of order, this cannot be directly detected). Since faults can in principle occur at anytime, it could even be the case that a new fault has appeared while we were attempting to restore the supply. However, to keep this benchmark manageable, we will assume that no new fault can occur during power restoration.

We let the reader elaborate on what are good choices for the next action to perform. If we choose to open SD10 and reclose CB1 and it works, this tends to favour hypothesis (B). In any case, this eliminates the possibility of a fault between CB1 and SD11, as well as the possibility of the actuator of SD11 lying in the context of hypotheses (A) or (C). If on the other hand CB1 opens again, we have to look further. Alternatively, we could also choose to test hypothesis (B) by closing SD62 and see whether CB6 opens, but this could be costly as this could lead to the temporary loss of a new feeder. Or perhaps we should close SD53 as in the original plan ... Complete examples will be found in the benchmark's website.

3.3 Main features of the problem

Three main features make this problem particularly interesting for state of the art planners. Firstly, partial observability is a crucial issue. Executing reconfiguration actions is necessary not only to change the state of the system, but also to gain vital information. Conversely, a good knowledge of the system's state is necessary to chose purposeful actions and avoid increasing breakdown costs. Unlike certain other problems involving partial observability, this one does not even offer the possibility of gaining information without taking intrusive actions and confronting the resulting observations with the expected ones. In sum, the problem is very representative of the need to trade off the gain of information which results from sensing and acting, the expected reward/penalty resulting from performing the right/wrong actions, and the cost of failing to act quickly. This is an optimisation problem, rather than one of merely reaching a desired state.

Secondly, the size of the state space and the structure of the problem make complete state enumeration – and a fortiori the incremental construction of a universal plan – absolutely impractical. At the same time, if plan utility is something to worry about, care should be exercised in pruning unlikely states in a belief state, as getting rid of an unlikely but potentially very costly state will spoil utility evaluation. Therefore, algorithms working with compact domain representations or using very effective domain control knowledge are necessary. Concerning designing appropriate control knowledge, a challenging aspect of the benchmark is that it is still an open problem: functions estimating the utility of network configurations exist, but the optimal solutions or rules of thumb for the

selection of actions with high utility are unknown. In fact, even discovering of how best to order given switching operations would be useful.

Finally, as in other planning benchmarks derived from real-world applications, see e.g., [11], actions have effects which are quite complex to model. For instance, closing a device increases the load of a circuit-breaker which should have the capacity to produce the additional resource. Closing a device may also lead to a fault being fed by the circuit-breaker, which will then open, leaving the lines on its feeder unsupplied. All this requires formalisms and planners which can handle domain constraints, infer ramifications and reason about resources, or at least enable the specification of elaborate context-dependent effects in the action descriptions.

4 Position of the problem in the literature

Supply restoration is not only very representative of issues that need to be addressed when dealing with real-world applications, but also of current research trends in planning and related areas. We now position the problem with respect to the literature, identify approaches which look likely to lead to advances in this domain, and describe existing work on this problem.

4.1 Planning and related areas

Progressive planners, such as TLplan [1] and TALplanner [12], appear as promising candidates to get somewhere with this benchmark in the very near future. PTLplan [10] is an extension of TLplan to deal with partial observability, stochastic actions, and probabilistic planning. Two features make progressive planners particularly interesting for the problem: the expressiveness of the planning language, and the extensive use of domain-control knowledge. After research effort is invested in understanding additional requirements placed by planning under uncertainty when it comes to the specification of domain control knowledge, these generic planners should be able to mimic strategies used by the domain specific supply restoration systems described below.

This said, progressive planners do not seem very-well equipped to deal with the full scope of the problem, and in particular with discovering plans with high utility. Firstly, there is no built-in capabilities for optimisation in TLplan yet. Secondly, because they perform explicit state enumeration, these planners are better suited to produce plans according to a domain-specific strategy, than to perform extensive search for optimal plans.

Recent planners working with much more compact domain representations include very expressive ones like Zander [13], which is based on stochastic satisfiability, and MBP [7], which is based on model-checking. At present, MBP generates plans that are guaranteed to achieve the goal despite sensor and actuator uncertainty. This is impractical for our benchmark where uncertainty just creates too many cases to be handled, and where gaining information up to the

point of being 100% sure to have reached a desired state often incurs unacceptable breakdown costs. Extensions to MBP which would relax this requirement would have a strong potential for excellent results on our problem.

In the longer term, the answer to the question of the production of high quality plans may well come from planners based on decision-theoretic regression [4]. First-order decision-theoretic regression is the key to symbolic dynamic programming, and does not require state or even action enumeration. It has been integrated with the situation calculus, leading to a very powerful problem-solving framework [5]. At present, this framework only deals with fully observable Markov decision processes, but once extensions to the partially observable case are available, our benchmark should be near-perfect to evaluate their benefits.

Another worthwhile direction would be to investigate the appropriateness for our problem of the general-purpose POMDP heuristics given in [6, 3].

Supply restoration is also typical of problems of interests in other areas such as model-based diagnosis, repair, reconfiguration, and execution, see e.g. [8, 15, 18]. The approaches in this area are often rely on a two level architecture featuring a diagnostic reasoner and a quasi-classical planner. If systems based on these approaches have so far been the most effective in dealing with similar application contexts, they still have some limits when confronted with our problem: they assume that all relevant information can be reliably acquired when needed, and that actions are reliable and sometimes elementary (a typical case is component replacements). Moreover, they are often applied to problems with belief states small enough for their content to be easily enumerated. These limits are largely due to the use of now obsolete planners which are unable to operate under uncertainty or to model actions with complex effects. Recent work on planning under uncertainty could be used in the framework of these approaches to remedy some of their current drawbacks.

A further line of research worth mentioning here is the modelling of diagnostic problem solving, including observations, actions, exogenous events, and diagnosis/repair/reconfiguration plans, in languages close to those used in planning, e.g. the situation calculus [14] or narratives [2]. It would be of interest to encode our example in these languages and experiment with the related planning technology [5, 12].

4.2 Existing work on the problem and related ones

Other works of interest are those concerned with similar AI applications to power systems. Space precludes more than the mention a few of them here.

SyDRe [16] is a simple decision-theoretic prototype for supply restoration on power distribution systems. It operates successfully in presence of an arbitrary number of faults, sensor and actuator uncertainties. However it does not reason on how to gain information: it generates a sequence of actions for the most probable state hypothesis, starts its execution, and revises this plan whenever the history of actions/observations shows that another hypothesis is more probable.

Diagnosis and supply restoration in power *transmission* systems has been studied e.g. in [8]. A crucial difference with our proposed testbed is that ob-

servations and actions are assumed to be reliable, which is reasonable when considering transmission systems. Sensor and actuator uncertainty make power distribution systems much more challenging.

The model-based reactive planner Burton has been applied to spacecraft engine reconfiguration [18]. Although this reconfiguration problem seems easier to handle – in particular observations and actions are reliable – it shares many aspects with the present benchmark. Indeed power distribution systems are another representative of the sensor rich, embedded, reconfigurable systems, which Williams and Nayak have dubbed *immobots* [17]. A number of choices made in Burton and SyDRe are similar. For instance the "upstream progression heuristic" is used in both, and they both sacrifice optimality for the sake of efficiency, by generating a sequence of actions for the most probable hypothesis and revising the plan if necessary.

The above systems can be used as a reference to measure the performance in time and solution quality of today's generic uncertainty planners. Ideally, we would like to see generic planners achieving comparable time performance by using extensive domain-control knowledge, as well as planners producing plans of much higher quality by reasoning on how to gain information.

5 Web site for this benchmark

We plan to make the items listed below available by early 2002 on the benchmark's web site `http://csl.anu.edu.au/~thiebaux/benchmarks/pds/`.

Formal description of the problem. For various reasons, the choice made in this paper is to provide a textual description of the problem rather than a formal one. We believe that this description is precise enough to be effectively usable. However, the web site will provide an extended version of the paper including a formal description in a TLplan-like language.

Network data and problem generator. Confidentiality issues prevent us to release data concerning existing EDF power distribution systems. However, we will make artificial data used in SyDRe's test suite available, including sample problems and suboptimal solutions produced by SyDRe. We also plan to provide a random network/problem generator for systematic experiments.

Simulator and supply restoration system. The Standard ML implementation of SyDRe will be downloadable from the web site. It includes a network simulator which can be used as a predictive model and a supply restoration component which can serve as a reference basis for comparative tests. However, since SyDRe employs a very miopic strategy which does not reason about how to reduce uncertainty, the ultimate goal is to obtain better quality plans than those produced by SyDRe.

6 Conclusion

This paper proposes the use of supply restoration in power distribution systems as a benchmark for planning under uncertainty. The time has come to mea-

sure planning systems against such realistic examples to complement the deeper analyses obtained with well-understood artificial problems. We have identified approaches which are likely candidates for progress with this benchmark, as well as their current limits. We hope that this paper will motivate the planning community to tackle the problem, and that the present decade will see success with at least a scaled down version of the above network example.

References

1. F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2), 2000.
2. C. Baral, S. McIlraith, and T. Son. Formulating diagnostic problem solving using an action language with narratives and sensing. In *Proc. KR*, 2000.
3. B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. AIPS*, pages 52–61, 2000.
4. C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.
5. C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *Proc. IJCAI*, 2001.
6. A.R. Cassandra, L. Kaelbling, and J.A Kurien. Acting under Uncertainty: Discrete Bayesian Models for Mobile-Robot Navigation. In *Proc. IROS-96*, 1996.
7. A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability. In *Proc. IJCAI*, 2001.
8. G. Friedrich and W. Nejdl. Choosing observations and actions in model-based diagnosis-repair systems. In *Proc. KR*, pages 489–498, 1992.
9. S. Hanks, M.E. Pollack, and P.R. Cohen. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 4(14):17–42, 1993.
10. L. Karlsson. Conditional progressive planning under uncertainty. In *Proc. IJCAI*, 2001.
11. J. Koehler and K. Schuster. Elevator control as a planning problem. In *Proc. AIPS*, pages 331–338, 2000.
12. J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and ressources. In *Proc. ECAI*, pages 501–505, 2000.
13. S.M. Majercik and M.L. Littman. Contingent planning under uncertainty via stochastic satisfiability. In *Proc. AAAI*, pages 549–556, 1999.
14. S. McIlraith. Explanatory diagnosis: conjecturing actions to explain observations. In *Proc. KR*, pages 167–177, 1998.
15. Y. Sun and D. Weld. Beyond simple observation: Planning to diagnose. In *Proc. AAAI*, pages 182–187, 1993.
16. S. Thiébaux, M.-O. Cordier, O. Jehl, and J.-P. Krivine. Supply restoration in power distribution systems — a case study in integrating model-based diagnosis and repair planning. In *Proc. UAI*, pages 525–532, 1996.
17. B. Williams and P. Nayak. Immobile robots – AI in the new millennium. *AI Magazine*, 17(3), 1996.
18. B. Williams and P. Nayak. A reactive planner for a model-based executive. In *Proc. IJCAI*, pages 1178–1185, 1997.

The DATA-CHASER and Citizen Explorer Benchmark Problem Sets

Barbara Engelhardt [1], Steve Chien [1], Anthony Barrett [1], Jason Willis [2], Colette Wilklow [2]

[1] Jet Propulsion Laboratory, California Institute of Technology
{firstname.lastname}@jpl.nasa.gov

[2] Previously at Space Grant Consortium, University of Colorado, currently at (1)

Abstract. This paper introduces two benchmark problem sets based on actual space mission operations. Each benchmark problem set includes problem generators, declarative specification of the problem(s), and one or more simulations. The first mission is the DATA-CHASER shuttle payload that flew onboard space shuttle Discovery flight STS-85 in 1997, and demonstrated the ability of automated mission planning to both reduce commanding effort and improve science return. The second mission is the Citizen Explorer Mission (CX-1), which is a small, earth orbiting satellite currently being prepared for launch. We include three problem classes of increasing complexity (and realism) for each mission scenario: planning and scheduling with states and resources (PSSR), PSSR with functional dependencies, and PSSR with functional dependencies and plan quality. The actual implementations are available for download from web sites at the University of Colorado, which designed and operated these spacecraft and missions.

1 Introduction

Historically, the research and applications communities in the area of automated planning and scheduling have not had significant amounts of interaction. As a consequence, there has not been significant transfer of information between the communities in either direction. Specifically, the cross-fertilization of communities is limited to a small number of research systems deployed in an ongoing operational context, and similarly only a few real-world planning and scheduling problems have breached the research community.

There are many reasons for this situation. It takes an incredible investment of time and energy for a researcher to learn the intricacies of an application domain. The research community and research institutions generally have not rewarded this investment of effort. Likewise, the community solving actual planning and scheduling problems did not have adequate incentive to work with the research community. With many difficult problems to solve in building functional systems, many of the central research areas are of lower priority. And in the commercial arena, there is significant negative incentive to distribute lessons painfully learned, which represent an important competitive advantage after all.

Fortunately, this situation appears to be changing. Within the research community, there is an increasing understanding of the importance of being relevant to the real world. With the appearance of startup companies and venture capital, the financial incentive to develop mature algorithms has grown considerably. Furthermore, with the

maturation of the field (and technology), the incentive for applied organizations to engage the research community has become more urgent.

This paper represents an effort to leverage the research community in developing techniques for integrated planning and scheduling problems that occur commonly for space mission operations. The remainder of this paper is organized as follows. First, we describe the basic elements that we provide for each testbed domain: a domain description, a declarative model, problem generators, and a simulation. For each such domain, we provide three versions of increasing complexity. A basic version of each domain includes planning and scheduling with resources. A more complex version adds functional dependencies. And the most complex version includes both functional dependencies and plan quality. Next, we describe each of the domains described in this paper: DATA-CHASER shuttle payload operations and Citizen Explorer (CX-1) satellite mission operations. For each domain we describe the background for the mission and mission goals. We then provide more details about the planning and scheduling problems. Next we describe the provided problem generators and simulators. Finally, we compare the problem domains presented with previously published domains in both the planning and scheduling and the operations research communities.

2 The Elements of a Testbed Domain

The first element of each domain is a textual description. This description gives the context of the model, problem generators, and simulation. It explains the mission being modeled and the overall problem context. It also references previous work in automated planning and scheduling solutions to the problem.

The second element of each domain is a model. This model is provided in the ASPEN Modeling Language (AML) [Sherwood et al. 1998]. AML is a mature representation language that has been used to represent planetary rover operations constraints [Sherwood et al. 2000] as well as space mission operations constraints for actual deployments [Smith et al. 2001, Wilkins&desJardins 2001]. While ideally these domain models could be provided in a more generic language, this format was chosen for two reasons. First, using AML facilitates timely release of the domain models by minimizing release effort. Given that it is desirable to release as many domain models in a timely fashion, it is hoped that others in the community will translate these models into a generic format. Second, certain aspects of the domain model would be difficult to represent in current generic domain description languages. The hope is that release of these models will spur extension of domain description languages. A description of the modeling language used for the original models is available for download from http://aspen.jpl.nasa.gov.

The third element of each problem domain is a problem generator. This is an executable (in these cases, Perl scripts) that can be used to generate a large number of initial states and goals for a planner to solve. In most cases the problem generator is parameterized to enable generating problems of varying size and difficulty.

The final element of each problem domain is a simulator. Once a planner has specified a plan, an execution simulation can be used to stochastically evaluate the effectiveness and the robustness of the plan for simulated missions operations. Effectiveness determines how well the planner satisfied the spacecraft goals, and can be measured by assessing the operational results, such as the science return, resource consumption, or state changes. Robustness, on the other hand, measures the ability of the planner to enable a successful mission in spite of significant run-time variations and anomalies, such as an action finishing early, consuming excessive resources, or

simply not executing correctly. Robustness can be measured by determining the number of inappropriate actions that are sent to the simulator, which in turn violate the constraints of the domain or put the spacecraft in an unsafe state.

The simulator itself has three parts: The database which stores the current state at time *t*, a set of specifications and constraints, and an executive, which receives action commands from the planner, attempts to execute them using the specification and constraint set, and updates the current state. The simulator is scalable so that there can be large or small simulations. The modeling of the simulator depends on how the planning language is defined, which determines whether activities are time-stamped, connected by constraints, or have conditional activities, or whether the planner can update its plan based on simulation feedback during the simulation.

For our domains, batch planners and continuous planners both use the same simulator. The planner is required to submit activities some number of seconds in advance of their scheduled execution, which is described as the *commit window*. The commit window must be greater than or equal to one second, and the planner must register the commit window length with the simulator when execution starts. The commit window size can change during execution, but the planner cannot modify activities once they have entered the commit window. The planner can receive updates from the simulator regarding activity parameter changes (such as start time or duration), state and resource updates, and current time. The simulator can warp so that the plans can run much faster than real time, relative to the commit window described by the particular planner. Batch planners can control the simulator either by setting the commit window to the duration of the plan (in which case the simulator can quickly warp through the entire simulation), or by passing parameterized activities at the appropriate times, but not replanning during the simulation. An important point to note here is that the domain information does not pose any restrictions on the use of planning technology to solve the problems. The planners could use constraint-based methods, committed search methods, or any other methods. Indeed, there is no restriction that a planner must be used, a smart executive or even arbitrary C code could be used to command the simulator. This opens the competition to truly test if planning technology is useful.

The stochastic model, or run-time variations, are stored as part of the specifications, which specify distributions instead of single values for certain variable features. Three different aspects of the mission can be impacted at run time: activity failure, resource consumption, and time and duration of state changes. Each domain has a stochastic and a non-stochastic simulator included in the release.

3 The DATA CHASER Mission

The DATA-CHASER was a Hitchhiker payload that flew onboard the Space Shuttle Discovery flight STS-85 in August 1997. (Figure 1) It had 3 co-aligned instruments that take data in the far and extreme ultraviolet wavelengths: far and extreme ultra-violet spectrometer (FARUS), soft x-ray and extreme ultraviolet experiment (SXEE), and a Lyman-Alpha solar imaging telescope (LASIT). In the actual DATA-CHASER mission, mission operations were automated using the DCAPS (DATA-CHASER Automated Planner and Scheduler) planning system [Chien et al. 1999]. The DATA-CHASER

Fig. 1. DATA-CHASER payload integrated into the STS-85 Shuttle Bay, STS-85 launching, and Payload Operator Jason Willis using DCAPS to command DATA-CHASER.

domain as modeled for the actual mission uses 67 resources and 58 activity types. Examples of resources include onboard power, a 4 MB memory buffer, and a 2 GB digital tape drive. Most of the systems have at least one state variable, which represents whether or not they are activated. Shuttle orientation is also modeled as a state variable. There are many concurrency resource constraints, for instance a downlink or uplink can only occur during contact with a TDRSS satellite. The activities include taking a picture with LASIT, changers for each state variable (such as opening/closing instrument doors), and descriptions of exogenous events like the shuttle passing to/from the Earth's shadow. Unfortunately, software integration difficulties before launch disabled part of the hardware during the mission. Our problems are based on the mission as originally designed.

DATA-CHASER problems involve trying to take observations within specified time windows given a number of exogenous events that change at different times. For instance, one consequence of flying on the shuttle system is that shuttle resources are shared and, hence, limited, with availability subject to change every 12 hours (the frequency at which NASA changes shuttle flight plans). These resources include access to uplink and downlink channels, and time that the payload is allowed to operate. Moreover, scientists would like to perform dynamic rescheduling during the mission. For instance, a solar flare can occur at random and drive a scientist's desire to rapidly alter the DATA-CHASER's activity schedule to reflect new requirements and goals, such as altered instrument priorities or longer integration times.

DATA-CHASER requires data and power management while gathering science. An automated scheduler searches for an optimal "data taking" schedule, while adhering to the constraints and resource restrictions. In its basic formulation, DATA-CHASER is a straightforward resource and state constrained scheduling problem that serves as a good introduction to the types of operations constraints common in spacecraft operations. A more complicated formulation requires representation of a number of functional dependencies including thermal and power constraints. In the full-blown formulation, DATA-CHASER represents a complex scheduling problem involving deadlines, observation windows, science preferences, linked observations, and engineering optimization criteria such as minimizing tape starts and stops as well as instrument door operations. There is no substantive planning (e.g. subgoaling) in the DATA-CHASER domain.

3.1 The DATA-CHASER Models and Problem Generators

The simplest DATA-CHASER model has over 46 activity types that are defined in terms of their effects on 19 resources and 9 states, which collectively represent the DATA-CHASER's external environment and subsystems. Such resources and states include the memory buffer, available power, communications availability periods, subsystem modes, and shuttle orientation. Most activities are possible commands to payload subsystems like performing an observation, moving data to a DAT recorder, or downlinking data. A smaller set of activities is for representing uncontrollable exogenous events like a shuttle orientation shift or entering a communications availability period. The five types of exogenous events to schedule around include:

- Shuttle orientation: The shuttle can point its cargo bay in one of four directions: Earth, Sun, Moon, and Deep Space. Given that the DATA-CHASER is a low priority Hitchhiker payload, it has no control over the orientation.
- Shuttle contamination: Occasionally the shuttle needs to fire its maneuvering rockets for orbit maintenance. In addition to accelerating the shuttle, this activity

contaminates local space for a short time. The DATA-CHASER has to close its main canister door during this time to keep its optics clean.

- Low data rate communications windows: During most of the mission the shuttle can provide a 1200 byte/sec downlink through the TDRSS satellite network, but a one to ten minute window exists in each orbit when no TDRSS satellite is in view.
- Medium data rate communications windows: Occasionally the mission will have a 25000 byte/sec downlink to a ground station, but availability depends on ground station visibility and the needs of other more important missions.
- Eclipse events: Once every orbit the shuttle travels through the Earth's shadow, and no solar observations are possible.

DCAPS-RES is our simplest DATA-CHASER planning model and illustrates planning with resources. The objective is to perform observations when the shuttle is not in the Earth's shadow, the cargo bay is facing the sun, and the shuttle has not recently contaminated the space around it. FARUS, SXEE and LASIT respectively take 72, 181, and 52 seconds and generate 5120 bytes, 48 bytes, and 2 megabytes per observation, and whole system generates a kilobyte of engineering telemetry per hour. Given that the memory buffer only has 4 MB, it is the most constrained resource. Since data can be rapidly transferred to the 2GB DAT recorder, there are naïve approaches to scheduling the observations by simply transferring the data as soon as it collected, but data on the DAT cannot be downlinked for rapid analysis during the mission. Rapid analysis is desired to let scientists alter the priorities of different observations to improve data quality. Thus some goals have explicit downlink requirements, making the scheduling problem slightly more difficult.

While our first model had fairly simple actions that took constant amounts of time and had static effects, our second model (called *DCAPS-PARM*) is slightly more complex in that it uses parameter dependency functions capture the context dependent thermal management problem. Since DATA-CHASER was mounted on a poorly conducting trellis in the vacuum of space, the only way to dissipate heat was through radiation. This means that the payload warmed when the sun beat down on it, and cooled during eclipses and when it pointed at deep space. We model this in terms of the temperature of the payload changing at rates determined by the shuttle's orientation, whether or not the canister door is open, and the power requirements of current activities. Given our model of heat, a schedule has a conflict due to calibration loss whenever the temperature falls outside of an 18° to 22° Celsius range.

Given *DCAPS-PARM*, we define *DCAPS-OPT* as an even harder third model to optimize science collection during a 12 hour period where different observations have context dependent payoffs depending on varying solar activity – an exogenous event.

The problem generators create random start states with subsequent 12-hour exogenous event scenarios and either a requested collection of observations or an observation payoff metric for the *DCAPS-OPT* model. We describe the exogenous events either in terms of cycles that start at a random point or markov models where time to take a transition is uniformly distributed between an upper and lower bound. For instance, the shuttle will be at some random point in its orbit and the day/night transition is a cycle starting at that point. Shuttle orientation provides an example of a markov model where scheduling to satisfy other payload needs results in changing the shuttle's orientation in random ways. In order to inject some realism into our markov-model-based exogenous events, we built our markov models from the 12-hour shuttle event sequences used during actual DATA-CHASER operations.

3.2 The DATA-CHASER Simulator

We evaluate solution plans for a problem by simulating them. The simulator takes exogenous events and grounded activities and determines what happens to the payload. For instance, having the contamination event with the CHASER door open may result in the instruments failing due to dirty optics. To evaluate solutions in each of the three models, the simulator has a flag to control the temperature component. For the simplest model, the simulator holds the temperature constant, and for the other models the simulator lets the temperature vary.

To make the problem more realistic, the simulator has a second flag to control a stochastic element. While actions in planning domains have explicit durations and effects, actions in reality have results that vary stochastically. For instance, a model might pessimistically state that it takes two minutes to transfer a LASIT image to the DAT recorder, but the actual time might vary from 100 to 120 seconds. In addition to time four other effects can vary around nominal operations:

- Datatakes: Datatakes will fail randomly 9% of the time. Actual power usage will differ from predicted power usage based on a normal distribution with a small variance. Failure rates and variances increase as the instruments' temperature approach the 18° and 22° Celsius bounds due to calibration problems.
- DAT Transfers: Transferring data to the DAT fails 2% of the time with data loss.
- Communications: Communications windows can drift slightly due to small variances between the Shuttle's orbit and the TDRSS satellite network.
- Thermal: The payload's rate of temperature change can vary by up to 5%.

Once given a problem description, the simulator takes grounded activities some set time in advance of executing them and giving sensory feedback in the form of failure notifications and the actual changes to the payload, which can stochastically different from the modeled expectations. This approach facilitates being able to test both batch planning approaches as well as incremental approaches. Upon completing the plan the simulator returns the plan's resulting score based on multiple criteria:

- The number of violated operational conflicts
- The number of successfully downlinked observations by observation type
- The number of observations stored on the DAT by observation type
- The number of times that the CHASER door opens and closes
- The number of activity failures by activity
- The total amount of power used while performing the observations
- An observation time based utility function for *DCAPS-OPT* problems.

4 The Citizen Explorer 1 (CX-1) Mission

The CX-1 spacecraft is a student designed and built spacecraft (Figure 2), developed by the Colorado Space Grant Consortium [Willis et.al. 1999]. The CX-1 satellite has a gravity gradient boom to keep it pointed to the Earth and uses two instruments to make atmospheric observations: a photometer to measure visible light intensity near the 365nm wavelength and a spectrophotometer to measure light intensities at different wavelengths between 280 and 350nm. These instruments will be used to perform atmospheric and climatological science coordinated with ground-based ozone and aerosol measurement. CX-1 was originally scheduled to launch on November 17, 2000. However, due to software and hardware difficulties with the communications subsystem, this launch has been delayed. Future launch possibilities are currently being negotiated. When launched, CX-1 will be in a sun-synchronous orbit around the

Fig. 2. The CX-1 Spacecraft, undergoing integration and test, and an artist's depiction of the CX-1 Spacecraft deployed and in flight

Earth at an altitude of 705 kilometers. The main ground tracking station will be in Colorado, and will downlink stored satellite data, provide real time health and status data, and uplink commands and control directives. CX-1 will also broadcast science and engineering data at UHF frequencies to schools at locations in the US.

CX-1 mission planning scenarios focus on the problem of acquiring the appropriate atmospheric measurements, downlinking the data to the correct schools during the upcoming pass, uplinking real-time activity requests from the University of Colorado, and monitoring spacecraft health and orbit patterns. Multiple constraints make this problem difficult. First, small data buffer sizes make CX-1 planning a highly resource constrained scheduling problem. Second, the time windows when CX-1 can downlink to a ground station are extremely limited by the limited onboard power and the small size of K-12 school ground stations (due to cost constraints),. Third, a large amount of data will be requested from the spacecraft so that it is important to optimize use of available downlink. Fourth, measurements made by CX-1 are categorized and driven by sun levels, hence the operations will vary based on near real-time feedback. While CX-1 operations does involve a small amount of planning (subgoaling), it is primarily a scheduling problem.

4.1 The CX-1 Models and Problem Generators

The model for CX-1 has approximately thirty-two activities, including data-takes, data downlinks, data uplinks, and engineering activities. These activities are defined in terms of their effects on thirteen resources and eight states, which collectively represent the CX-1's subsystems and external environment. These resources and states include the flash memory buffer, communications link, available battery power, solar array power, a sun sensor, ground station in-view periods, climate modes, and transmitter modes. Activities are either commands to satellite subsystems or exogenous events affecting the satellite's states or resources, such as entering/exiting the view of a ground station or entering/exiting direct sunlight.

The model also has mission and operations constraints, which can impact activities (temporal constraints), resources, or states. Temporal constraints include requirements that a series of critical housekeeping operations must be performed at the transitions between light and dark. Resource constraints include requirements that the flash memory buffer cannot exceed its capacities and that the battery power cannot go below half of its maximum capacity. State constraints include requirements that uplinks and downlinks only occur when the Colorado station in in-view and that the transmitter is in transmit mode. Also, school downlinks (which do not move any memory off of the flash memory buffer) can only occur when the schools are in-view and the transmitter is in broadcast mode, engineering activities must take place when the sun is not directly visible, and data takes must occur while the sun is in-view.

CX-1 problems involve obtaining and downlinking the maximum amount of atmospheric data while staying inside power and memory resource guidelines. The in-view duration for the Colorado ground station varies depending on the orbit of the satellite and outages in the ground station. Removing data from the flash memory buffer before it fills requires downlinking as much as possible during each visibility window, and this requires dynamic rescheduling as viewing periods change. Also,

changes in power consumption by certain activities affects the power profile and can drain excessive power from the satellite. Here dynamic rescheduling facilitates discarding activities to stay within power guidelines.

There are three different CX-1 models with increasing complexity. In *CX1-RES*, activities have constant durations and have constant resource and state needs. This model represents a straightforward space mission operation domain for planning and scheduling with constrained states and resources. The *CX1-PARM* model adds a number of parameter dependencies. For some activities, power usage is a function on activity duration and lighting state, and takes as much power as possible from the solar power source before relying on battery power. Battery usage is a function of current levels of battery charge and the duration of the activity. Downlinking bandwidth is a function of the modes of the satellite. Finally, the *CX1-OPT* model further increases the complexity by including optimality criteria. These criteria are based on the total amount of data downlinked to the ground stations (preferring more data), the largest amount of data contained in the memory buffer at any one time (with a preference for less), the amount of data acquired while in non-tropical climates (preferring more), the number of mode switches in the transmitter, and the smallest charge on the available power (preferring a higher minimum charge).

With these models the CX-1 problem generators create six different files containing activity instantiations for a user specified number of orbits. Some of these activities are changers for exogenous events and they cannot be deleted, removed, or modified in the schedule. Others are requests for downlinking, engineering events, and datatakes, and can be modified, added to, or removed in the final schedule.

- Initialization Activities: In the start state, there is a random amount of power in the battery and a random amount of data currently stored in the flash memory.
- Engineering Data Requests: Engineering scans are requested approximately every 280 seconds while the spacecraft is in both in-view of the sun and in darkness.
- Datatake Requests: These requests occur approximately every 170 seconds when the spacecraft is in-view of the sun.
- Sun In-View Periods: These periods are generated randomly based on an estimate of a CX-1 proposed orbit and a somewhat random start position. The spacecraft always begins the simulator in darkness. The climate (tropical or non-tropical) transitions are generated relative to the sun in-view periods.
- Power Activities: Solar power activities add a random amount of power to the available power resource, and occur approximately every 300 seconds while the spacecraft is in-view of the sun.
- Downlink Station In-View Periods: Downlink windows to both the Colorado ground station and participating schools are generated randomly based on viewing windows for an estimated orbit and sun in-view periods. Each window's duration is based on both satellite position and the strength of the receiving station.

4.2 CX-1 Simulation

We can describe the mission operations of the CX-1 model, as described above, in terms of the stochastic element of the planning simulation in order to illustrate how the simulation can differ from predicted (or nominal) operations. For instance, many activities have actual power usages that are normally distributed with a small variance around their predicted power usages.

- Engineering Scans: Engineering scans fail randomly 7% of the time.
- Datatakes: Datatakes will fail randomly 9% of the time.

- Solar Power Functions: The actual amount of solar power is normally distributed around the model's predicted solar power with a small variance.
- Light/Dark Cycle: The sunlight/darkness cycle may be shifted based on a cyclic model of satellite drift. The shift also impacts entering and exiting tropical zones.
- Downlinking: Satellite drift impacts all downlink opportunity windows. The start time may be moved (i.e., delayed lock up) and the duration may be shortened (i.e., signal cut off) based on a cyclic model of satellite drift.
- Bandwidth: The bandwidth to downlink both spacecraft data and science data is a uniformly distributed number, owing to possible downlink problems.

These stochastic parameters can remain ungrounded until run-time where they impact the planner's effectiveness. Many runs can be performed on the simulator to estimate the expected performance of the planner. For testing purposes, we also include a "happy simulator" which runs nominal operations with zero variance.

The CX-1 simulator receives parameterized activities from the planner and simulates operations for the entire duration of the plan. Throughout the the plan's execution, the simulator calculates the score based on the following criteria:

- Number of violated operational conflicts (e.g., dual usage of atomic resources, overflowing the memory buffer, or downlinks to nonexistent ground stations)
- Total amount of data downlinked to the Colorado ground station
- Flash memory usage (preferring a lower mean usage and smaller variance)
- Available power (preferring a higher mean and smaller variance)
- Total amount of data downlinked to the school ground stations
- Total amount of data acquired while in a non-tropical zone
- Total number of transmitter mode switches

5 Accessing the Problem Set Information

The problem set information is downloadable from the University of Colorado Space Grant Web Site (www-sgc.colorado.edu). While the release sets are preliminary and are still undergoing slight revisions (and testing), they are very close to the final sets. This site will also be the focal point for future updates and releases. While the domains are being made available to the public, specific terms are listed on the web site – including acknowledgement of the source in the event of any usage of the material, prohibitions on commercial use, etc.

6 Discusion: Future Work, Related Work, Conclusions

We hope that these domains will be the first in a series of space mission operations domains released for use by the automated planning and scheduling community. As we have less mature collaboration efforts with three other University Nanosatellite projects, we hope that eventually we will be able to release similar problem sets relating to these. An important aspect of this work involves determining standards for releasing domains – i.e. a formalization of the domain, problem generator, and simulator specifications. Developing a sufficiently expressive domain representation standard would facilitate use of the problem sets by other research groups.

While there has been a historical disconnect between the research and applications oriented planning communities, a number of planning-oriented domains and testbeds have been made available. These testbeds have originated in the research community by experimenters as they either improve an existing planner's performance or define

algorithms that plan with ever more expressive action representations. For instance, the classical PRODIGY and UCPOP planners had multiple test domains included in their releases. The sensory GRAPHplan package [Graphplan] has accompanying domains as well. Our testbeds differ from this work due to our underlying focus on real world problems instead of planner test cases.

On the planner comparison side, many planner optimization papers report planner performance on a number of benchmark problems, and the most realistic of these is a logistics problem to move packages around an artificial map. Additionally, the AIPS 98 [McDermott 2000] and 2000 [Bacchus] planning competitions had problems defined in a standard modeling language called PDDL, and test domains with problem generators and plan simulators were used. Our domains are different than these previously released testbeds in that they are derived from actual space mission operations problems and address integrated planning and scheduling with metric time, resources, functional dependencies, and optimization (although a number of AIPS 2000 domains had some of these elements).

The part-machining domain [Gil 1991] and the elevator control domain [Koehler & Schuster 2000] were motivated by real problems. The part machining was an attempt to extract domain knowledge about how to turn a mass of metal into a machined part and encode it into a PRODIGY domain. Similarly, the elevator control domain involved taking a set of services and constraints and encoding them in PDDL to be solved by planners such as those participating in the AIPS competitions. One of the results from these efforts involved determining where the established modeling language cannot represent a desired feature of the real world problem. For instance, the elevator control problem has a capacity constraint that PDDL could not represent. Our work differs from this research in two places. We do not avoid time and other metric constraints, and we altered our simulators to facilitate experimenting with interleaved planning and execution.

A number of additional pure scheduling benchmarks exist [Fox & Ringer] as well as makespan benchmarks. These are designed to be more manufacturing and enhanced job-shop scheduling problems. In contrast, our work emphasizes the integrated planning and scheduling inherent in space mission operations.

Other research on interleaved planning and execution has resulted in shared testbeds like tileworld, truckworld, and the phoenix testbeds [Hanks et al. 1993]. These worlds were generated as benchmark cases for agent design within a multi-agent context. While tileworld and truckworld were relatively simplistic testbeds for testing agent systems, the phoenix testbed focused on a forest firefighting domain. Each of these testbeds offered defining problems with varying complexity, but only the phoenix testbed had an underlying operations scenario like our testbeds. The Robocup rescue project [Kitano et al 2001] also focuses on providing testbeds with an underlying operations scenario. It targets distribution of a complex multi-agent simulation environment. This environment has the potential to provide an extremely rich planning and scheduling testbed.

This paper introduced two benchmark problem sets based on actual space mission operations. Each benchmark problem set includes problem generators, declarative specification of the problem(s), and one or more simulations. The first mission is the DATA-CHASER, which demonstrated the ability of automated mission planning to both reduce commanding effort and improve science return [Chien et al. 1999]. The second mission is the Citizen Explorer Mission (CX-1), which is currently being rescheduled for launch. We include three problem classes of increasing complexity (and realism): planning and scheduling with states and resources (PSSR), PSSR with functional dependencies, and PSSR with functional dependencies and plan quality.

The domain descriptions, problem generators, and simulators are available for download from a web site at the University of Colorado, which designed and built these spacecraft and missions (and operated DATA-CHASER). It is our hope that release of this information will help to focus the planning and scheduling research community on key issues in planning and scheduling including: domain model expressiveness, representing functional dependencies, and plan optimization.

7 References

1. Sherwood, R., et al. "Using ASPEN to Automate EO-1 Activity Planning," Proceedings 1998 IEEE Aerospace Conference, Aspen, Colorado, March, 1998.
2. Sherwood, R., Estlin, T., Chien, S., Rabideau, F., Engelhardt, B., Mishkin, A., and Cooper, B., "An Automated Rover Command Generation Prototype for the Mars 2001 Marie Curie Rover," SpaceOps 2000, Toulouse, France, June 2000.
3. Smith, B.D., Engelhardt, B.E., Mutz, D., "Automated Mission Planning for the Modified Antarctic Mapping Mission," Proceedings 2001 IEEE Aerospace Conference, Big Sky, Colorado, March, 2001.
4. Wilkins, D. and desJardins, M., "A Call for Knowledge-based Planning," AI Magazine, 11(1): 99-115, 2001.
5. Chien, S., Rabideau, G., Willis, J., and Mann, T., "Automating Planning and Scheduling of Shuttle Payload Operations," Artificial Intelligence Journal 114 (1999) 239-255.
6. Willis, J., Rabideau, G., Wilklow, C., "The Citizen Explorer Scheduling System," Proceedings of the IEEE Aerospace Conference, Aspen, CO, March 1999.
7. Sensory GraphPlan home page, http://www.cs.washington.edu/ai/sgp .html
8. McDermott, D. The 1998 AI Planning Systems Comp. AI Mag 21(2), 2000.
9. Bacchus, F. The AIPS-00 Planning Competition. http://www.cs.toronto.edu/aips2000
10. Gil, Y., "*A Specification of Manufacturing Processes for Planning,*" CMU-CS-91-179.
11. Koehler, J. and Schuster, K., "Elevator Control as a Planning Problem," Proc 5th Intl Conf on Art Intelligence Planning Systems, Breckenridge, CO. April, 2000.
12. B. Fox and M. Ringer, Resource Constrained Scheduling Problem Home Page, http://www.neosoft.com/~benchmrx/
13. S. Hanks, M. E. Pollack, and P.Cohen, "Benchmarks, Testbeds, Controlled Experimentation, and the Design of Agent Architectures, AI Magazine, 14(4):17-42, 1993. (also see http://www.cs.pitt.edu/~pollack/distrib/tileworld.html)
14. H. Kitano and S. Tadakoro, "RoboCup Rescue: A Grand Challenge for Multiagent and Intelligent Systems," AI Magazine, 11(1): 39-52, 2001. (also see http://www.r.cs.kobe-u.ac.jp/robocup-rescue/)

September 13

Sapa: A Domain-Independent Heuristic Metric Temporal Planner

Minh B. Do & Subbarao Kambhampati *
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
{binhminh,rao}@asu.edu
http://rakaposhi.eas.asu.edu/sapa.html

Abstract

Many real world planning problems require goals with deadlines and durative actions that consume resources. In this paper, we present *Sapa*, a domain-independent heuristic forward chaining planner that can handle durative actions, metric resource constraints, and deadline goals. The main innovation of *Sapa* is the set of distance based heuristics it employs to control its search. We consider both optimizing and satisficing search. For the former, we identify admissible heuristics for objective functions based on makespan and slack. For satisficing search, our heuristics are aimed at scalability with reasonable plan quality. Our heuristics are derived from the "relaxed temporal planning graph" structure, which is a generalization of planning graphs to temporal domains. We also provide techniques for adjusting the heuristic values to account for resource constraints. Our experimental results indicate that *Sapa* returns good quality solutions for complex planning problems in reasonable time.

1 Introduction

For most real world planning problems, the STRIPS model of classical planning with instantaneous actions is inadequate. We normally need plans with durative actions that execute concurrently. Moreover, actions may consume resources and the plans may need to achieve goals within given deadlines. While there have been efforts aimed at building metric temporal planners that can handle different types of constraints beyond the classical planning specifications [14, 9, 11], most such planners either scale up poorly or need hand-coded domain control knowledge to guide their search. The biggest problem faced by existing temporal planners is thus the control of search (c.f. [17]). Accordingly, in this paper, we address the issues of domain independent heuristic control for metric temporal planners.

At first blush search control for metric temporal planners would seem to be a very simple matter of adapting the work in heuristic planners in classical planning [3, 12, 7]. The adaptation however does pose several challenges. To begin with, metric temporal planners tend to have significantly larger search spaces than classical planners. After all, the problem of planning in the presence of durative actions and metric resources subsumes both the classical planning and scheduling problems. Secondly, the objective of planning may not be limited to simple goal satisfaction, and may also include optimization of the associated schedule (such as maximum lateness, weighted tardiness, weighted completion time, resource consumption etc. [15]). Finally, the presence of metric and temporal constraints, in addition to subgoal interactions, opens up many more potential avenues for extracting heuristics (based on problem relaxation). Thus, the question of which relaxations provide best heuristics has to be carefully investigated.

In this paper, we present *Sapa*, a heuristic metric temporal planner that we are currently developing. *Sapa* is a forward chaining metric temporal planner, whose basic search routines are adapted from Bacchus and Ady's[1] recent work on temporal TLPlan. We consider a forward chaining planner because of the advantages offered by the complete state information in handling metric resources [17]. Unlike temporal TLPlan, which relies on hand-coded control knowledge to guide the planner, the primary focus of our work is on developing distance based heuristics to guide the search. In *Sapa*, we estimate the heuristic values by doing a phased relaxation: we first derived heuristics from a relaxation that ignores the delete effects and metric resource constraints, and then adjust these heuristics to better account for resource constraints. In the first phase, we use a generalization of the planning graphs [2], called relaxed temporal planning graphs (RTPG), as the basis for deriving the heuristics. Our use of planning graphs is inspired by (and can be seen as an adaptation of) the recent work on *AltAlt* [12] and FF [7]. We consider both optimizing and satisficing search scenarios. For the former, we develop admissible heuristics for objective functions based on makespan or slack. For the latter, we develop very effective heuristics that use the characteristics of a "relaxed" plan

*We thank David E. Smith, Terry Zimmerman and three anonymous reviewers for useful comments on the earlier drafts of this paper. This research is supported in part by the NSF grant IRI-9801676, and the NASA grants NAG2-1461 and NCC-1225.

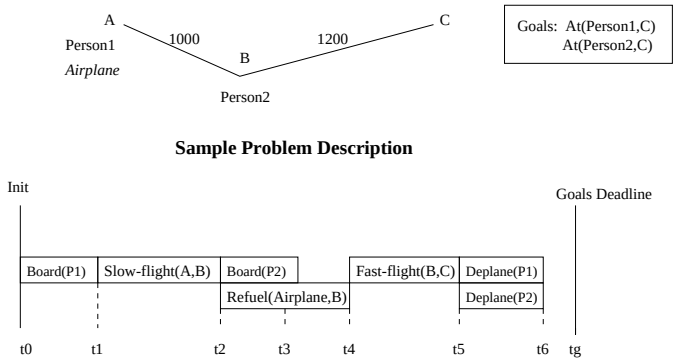


Figure 1: Sample problem description and its solution

derived from the planning graphs. Finally, we present a way of improving the informedness of our heuristics by accounting for the resource constraints (which are ignored in constructing the relaxed planning graphs).

Sapa is implemented in Java. Our empirical studies indicate that *Sapa* can solve planning problems with complex temporal and resource constraints quite efficiently. *Sapa* also returns good quality solutions with short makespans and very few ir relevant actions. This is particularly encouraging given that temporal TLPlan, the leading contender of *Sapa* that uses hand-coded control knowledge, tends to output many irrelevant actions.

The rest of this paper describes the development and evaluation of *Sapa.* We start in Section 2 with a discussion of action representation and the general search algorithm used in *Sapa.* In Section 3, we present the relaxed planning graph structure and discuss different heuristics extracted from it. We also describe how to adjust the heuristic values based on the metric resource constraints. We present empirical results in Section 4 and conclude the paper with a discussion of related work and future work in Sections 5 and 6.

2 Handling concurrent actions in a forward state space planner

Sapa addresses planning problems that involve durative actions, metric resources, and deadline goals. In this section, we describe how such planning problems are represented and solved in *Sapa.* We will first describe the action representation, and will then present the forward chaining state search algorithm used by *Sapa.*

To illustrate the representation and the search algorithm used in *Sapa,* we will use a small example from the flying domain discussed in [14]. In this domain, which we call *zeno-flying*, airplanes move passengers between cities. An airplane can choose between "slow flying" and "fast flying" actions. "Slow flying" travels at 400 miles/hr and consumes 1 gallon of fuel for every 3 miles. "Fast flying" travels at 600 miles/hr and consumes 1 gallon of fuel every 2 miles. Passengers can be *boarded* in 30 minutes and *deplaned* in 20 minutes. The fuel capacity of the airplane is 750 gallons and it takes 60 minutes to *refuel* it. Figure 1 shows a simple problem from this domain that we will use as a running example throughout the paper. In this problem, Person1 and the Airplane are at cityA, Person2 is at cityB and the plane has 500 gallons of fuel in the initial state. The goals are to get both Person1 and Person2 to cityC in 6.5 hours. One solution for this problem, shown in the lower half of Figure 1, involves first *boarding* Person1 at cityA, and then *slow-flying* to cityB. While *boarding* Person2 at cityB, we can *refuel* the plane concurrently. After finishing refueling, the plane will have enough fuel to *fast-fly* to cityC and *deplane* the two passengers.

2.1 Action representation

Planning is the problem of finding a set of actions and their respective execution times to satisfy all causal, metric, and resource constraints. Therefore, action representation has influences on the representation of the plans and on the planning algorithm. In this section, we will discuss the action representation used in *Sapa.* Our representation is influenced by the PDDL+ language proposal[4] and the representations used in Zeno[14] and LPSAT[19] planners.

Unlike actions in classical planning, in planning problems with temporal and resource constraints, actions are not instantaneous but have durations. Their preconditions may either be instantaneous or durative and their effects may occur at any time point during their execution. Each action *A* has a duration D_A, starting time S_A, and end time E_A (= $S_A + D_A$). The value of D_A can be statically defined for a domain, statically defined for a particular planning problem, or can be dynamically decided at the time of execution.[1] Action *A* have preconditions $Pre(A)$ that may

[1]For example, in the zeno-flying domain discussed earlier, we can decide that boarding a passenger always takes 10 minutes for all problems in this domain. Duration of the action of flying an airplane between two cities will depend on the distance between these two cities. Because the distance between two cities will not change over time, the duration of a particular flying action will be totally specified once we parse the planning

```
(:action BOARD
    :parameters
    (?person - person ?airplane - plane ?city - city)
    :duration (st, + st 30)
    :precondition
    (and (at ?person ?city) - (st,st)
         (in-city ?airplane ?city) - (st,et))
    :effect
    (and (not (at ?person ?city)) - st
         (in ?person ?airplane) - et))
```

```
(:action SLOW-FLYING
    :parameters
    (?airplane - plane ?city1 - city ?city2 - city)
    :duration
    (st, + st (/ (distance ?city1 ?city2)
                 (slow-speed ?airplane)))
    :precondition
    (and (in-city ?airplane ?city1) - (st,st)
         (> (fuel ?airplane) 0) - (st,et))
    :effect
    (and (not (in-city ?airplane ?city1)) - st
         (in-city ?airplane ?city2) - et
         (-= (fuel ?airplane)
            (* #t (sf-fuel-cons-rate ?airplane))) - #t))
```

Figure 2: Examples of action descriptions in *Sapa*

be required either to be instantaneously true at the time point S_A, or required to be true starting at S_A and remain true for some duration $d \leq D_A$. The logical effects *Eff(A)* of *A* will be divided into three sets $E_s(A)$, $E_e(A)$, and $E_m(A, d)$ containing respectively instantaneous effects at time points S_A, E_A and $S_A + d$ ($0 < d < D_A$). Figure 2 illustrates the actual representations used in *Sapa* for actions *boarding* and *slow-flying* in the zeno-flying domain. Here, *st* and *et* denote the starting and ending time points of an action, while *#t* represents a time instant between *st* and *et*. While the action $boarding(person, airplane, city)$ requires a person to be at the location *city* only at its starting time point *st*, it requires an airplane to stay there the duration of its execution. This action causes an instant effect $(not(at(?person, ?city)))$ at the starting time point *st* and the delayed effect $in(?person, ?airplane)$ at the ending time point *et*.

Actions can also consume or produce metric resources and their preconditions may also well depend on the value of the corresponding resource. For resource related preconditions, we allow several types of equality or inequality checking including ==, <, >, <=, >=. For resource-related effects, we allow the following types of change (update): assignment(=), increment(+=), decrement(-=), multiplication(*=), and division(/=). In Figure 2, the action *slow-flying* requires the fuel level to be greater than zero over the entire duration of execution and consumes the fuel at a constant rate while executing.

Currently we only model and test domains in which effects occur at the start or end time points, and preconditions are required to be true at the starting point or should hold true throughout the duration of that action. Nevertheless, the search algorithm and the domain representation schema used in *Sapa* are general enough to represent and handle actions with effects occurring at any time point during their durations and preconditions that are required to hold true for any arbitrary duration between the start and end time points of an action. In the near future, we intend to test our planner in domains that have more flexible temporal constraints on the preconditions and effects of actions.

2.2 A forward chaining search algorithm

Even though variations of the action representation scheme described in the previous section have been used in the partial order temporal planners such as IxTeT[9] and Zeno[14] before, Bacchus and Ady [1] are the first to propose a forward chaining algorithm capable of using this type of action representation and allow concurrent execution of actions in the plan. We adapt their search algorithm in *Sapa*.

Before going into the details of the search algorithm, we need to describe some major data structures that are used. *Sapa*'s search is conducted through the space of time stamped states. We define a time stamped state S as a tuple $S = (P, M, \Pi, Q, t)$ consisting of the following structure:

- $P = (\langle p_i, t_i \rangle \mid t_i < t)$ is a set of predicates p_i that are true at *t* and the last time instant t_i at which they are achieved.[2]
- *M* is a set of values of all functions representing all the metric-resources in the planning problem. Because the continuous values of resource levels may change over the course of planning, we use *functions* to represent the resource values.
- Π is a set of persistent conditions, such as action preconditions, that need to be protected during a period of time.
- *Q* is an event queue containing a set of updates each scheduled to occur at a specified time in the future. An event *e* can do one of three things: (1) change the True/False value of some predicate, (2) update the value of some function representing a metric-resource, or (3) end the persistence of some condition.

problem. However, *refueling* an airplane may have a duration that depends on the current fuel level of that airplane. We may only be able to calculate the duration of a given *refueling* action according to the fuel level at the exact time instant when we execute that action.

[2]For example, at time instant t_1 in Figure 1, $P = \{\langle At(airplane, A), t_0\rangle, \langle At(Person2, B), t_0\rangle, \langle In(Person1, t_1)\rangle\}$

- t is the time stamp of S

In this paper, unless noted otherwise, when we say "state" we mean a time stamped state. It should be obvious that time stamped states do not just describe world states (or snap shots of the world at a given point of time) as done in classical progression planners, but rather describe both the state of the world and the state of the planner's search.

The initial state S_{init} is stamped at time 0 and has an empty event queue and empty set of persistent conditions. However, it is completely specified in terms of function and predicate values. In contrast, the goals do not have to be totally specified. The goals are represented by a set of n 2-tuples $G = (\langle p_1, t_1\rangle ... \langle p_n, t_n\rangle)$ where p_i is the i^{th} goal and t_i is the time instant by which p_i needs to be achieved.

Goal Satisfaction: The state $S = (P, M, \Pi, Q, t)$ *subsumes* (entails) the goal *G* if for each $\langle p_i, t_i\rangle \in G$ either:

1. $\exists \langle p_i, t_j\rangle \in P$, $t_j < t_i$ and there is no event in *Q* that deletes p_i.
2. There is an event $e \in Q$ that adds p_i at time instant $t_e < t_i$.

Action Application: An action A is *applicable* in state $S = (P, M, \Pi, Q, t)$ if:

1. All instantaneous preconditions of *A* are satisfied by *P* and *M*.
2. *A*'s effects do not interfere with any persistent condition in Π and any event in *Q*.
3. No event in *Q* interferes with persistent preconditions of *A*.

When we apply an action *A* to a state $S = (P, M, \Pi, Q, t)$, all instantaneous effects of *A* will be immediately used to update the predicate list *P* and metric resources database *M* of *S*. *A*'s persistent preconditions and delayed effects will be put into the persistent condition set Π and event queue *Q* of *S*. For example, if we apply action *Board(P1,airplane)* to the initial state of our running example in Figure 1, then the components of resulting state *S* will become $P = \{\langle At(airplane, A), t_0\rangle, \langle In(P1, airplane), t_0\rangle, \langle At(P2, B), t_0\rangle\}$, *M* = {*Fuel(airplane)=500*}, $\Pi = \{\langle At(airplane, A), t_1\rangle\}$, and $Q = \{\langle In(P1, airplane), t_1\rangle\}$.

Besides the normal actions, we will have one special action called **advance-time**[3] which we use to advance the time stamp of *S* to the time instant t_e of the earliest event e in the event queue *Q* of *S*. The advance-time action will be applicable in any state *S* that has a non-empty event queue. Upon applying this action, we update state *S* according to all the events in the event queue that are scheduled to occur at t_e.

Notice that we do not consider action *A* to be applicable if it causes some event *e* that interferes with an event e' in the event queue, even if *e* and e' occur at different time points. We believe that even though an event has instant effect, there should be some underlying process that leads to that effect.[4] Therefore, we feel that if two actions cause instant events that are contradicting with each other, then even if the events occur at different time points, the underlying processes supporting these two events may contradict each other. Thus, these two actions are not allowed to execute concurrently. Our approach can be considered as having a *hold* process [5] extending from the starting point of an action to the time point at which an event occurs. The hold process protects that predicate from violations by conflicting events from other actions. This also means that even though an effect of a given action *A* appears to change the value of a predicate at a single time point *t*, we implicitly need a duration from the starting point *st* of *A* to *t* for it to happen. We are currently investigating approaches to represent constraints to protect a predicate or resource more explicitly and flexibly. Additionally, in handling metric resource interactions between two actions, *Sapa* follows an approach similar to the ones used by Zeno[14] and RIPP[8]: it does not allow two actions that access the same metric resource to overlap with each other. By not allowing two actions affecting the same resource to overlap, we can safely change the resource condition that needs to be preserved during an action to be an instantaneous condition or an update at the start or end point of that action. For example, the condition that the fuel level of an airplane should be higher than 0 while flying between two cities, can be changed to a check to see if the level of fuel it has at the beginning of the action is higher than the amount that will be consumed during the course of that action. This helps in simplifying the search algorithm. In future, we intend to investigate other ways to relax this type of resource interaction constraints.

Search algorithm: The basic algorithm for searching in the space of time stamped states is shown in Figure 3. We proceed by applying all applicable actions to the current state and put the result states into the sorted queue using the $Enqueue()$ function. The $Dequeue()$ function is used to take out the first state from the state queue. Currently, *Sapa* employs the A* search. Thus, the state queue is sorted according to some heuristic function that measures the difficulty of reaching the goals from the current state. The rest of the paper discusses the design of heuristic functions.

3 Heuristic control

For any type of planner to work well, it needs to be armed with good heuristics to guide the search in the right direction and to prune the bad branches early. Compared with heuristic forward chaining planners in classical planning, *Sapa* has many more branching possibilities. Thus, it is even more critical for *Sapa* to have good heuristic guidance.

[3] *Advance-time* is called **unqueue-event** in [1]

[4] For example, the *boarding* action will cause the event of the passenger being inside the plane at the end of that action. However, there is an underlying process of taking the passenger from the gate to inside the plane that we are not mentioning about.

State Queue: *SQ*={S_{init}}
while *SQ*≠{}
S:= *Dequeue*(*SQ*)
Nondeterministically select *A*
applicable in *S*
S' := *Apply*(*A*,*S*)
if *S'*|= *G* **then**
PrintSolution
else *Enqueue*(*S'*,*SQ*)
end while;

Figure 3: Main search algorithm

Heuristic	Objective Function	Basis	Adm.	Use res-infor
Max-span	minimize makespan	RTPG	Yes	No
Min-slack	maximize minimum slack	RTPG	Yes	No
Max-slack	maximize maximum slack	RTPG	Yes	No
Sum-slack	maximize sum-slack	RTPG	Yes	No
Sum-action	minimize number of actions	relaxed plan	No	No
Sum-duration	minimize sum of action durations	relaxed plan	No	No
Adj. sum-act.	minimize number of actions	relaxed plan	No	Yes
Adj. sum-dur.	minimize sum of action durations	relaxed plan	No	Yes

Table 1: Different heuristics investigated in *Sapa.* Columns titled "objective function", "basis", "adm" and "use res-infor" show respectively the objective function addressed by each heuristic, the basis to derive the heuristic values, the admissibility of the heuristic, and whether or not resource-related information is used in calculating the heuristic values.

Normally, the design of the heuristics depends on the objective function that we want to optimize; some heuristics may work well for a specific objective function but not others. In a classical planning scenario, where actions are instantaneous and do not consume resources, the quality metrics are limited to a mere count of actions or the parallel execution time of the plan. When we extend the classical planning framework to handle durative actions that may consume resources, the objective functions need to take into account other quality metrics such as the makespan, the amount of slack in the plan and the amount of resource consumption. Heuristics that focus on these richer objective functions will in effect be guiding both planning and scheduling aspects. Specifically, they need to control both action selection and the action execution time.[5]

In this paper, we consider both satisficing and optimizing search scenarios. In the former, our focus is on efficiently finding a reasonable quality plan. In the later, we are interested in the optimization of objective functions based on makespan, or slack values. We will develop heuristics for guiding both types of search. Table 1 provides a high level characterization of the different heuristics investigated in this paper, in terms of the objective functions that they are aimed at, and the knowledge used in deriving them.

For any type of objective function, heuristics are generally derived from relaxed problems, with the understanding that the more constraints we relax, the less informed the heuristic becomes [13]. Exploiting this insight to control a metric temporal planner brings up the question of what constraints to relax. In classical planning, the "relaxation" essentially involves ignoring precondition/effect interactions between actions [3, 7]. In metric-temporal planning, we can not only relax the logical interactions, but also the metric resource constraints, and temporal duration constraints.

In *Sapa*, we estimate the heuristic values by doing a phased relaxation: we first relax the delete effects and metric resource constraints to compute the heuristic values, and then modify these values to better account for resource constraints. In the first phase we use a generalization of the planning graphs [2], called relaxed temporal planning graphs (RTPG), as the basis for deriving the heuristics. Our use of planning graphs is inspired by (and can be seen as an adaptation of) the recent work on *AltAlt* [12] and FF [7]. The RTPG structures are described in Section 3.1, and Sections 3.2 and 3.3 describe the extraction of admissible and effective heuristics from the RTPG. Finally, in Section 3.4, we discuss a technique for improving the informedness of our heuristics by adjusting the heuristic values to account for the resource constraints (which are ignored in the RTPG).

3.1 Building the relaxed temporal planning graph

All our heuristics are based on the relaxed temporal planning graph structure (RTPG). This is a Graphplan-style[2] bi-level planning graph generalized to temporal domains. Given a state $S = (P, M, \Pi, Q, t)$, the RTPG is built from

[5] In [17], Smith et. al. discuss the importance of the choice of actions as well as the ordering between them in solving complicated real world planning problems involving temporal and resource constraints.

```
while(true)
    forall A ≠advance-time applicable in S
            S := Apply(A,S)
            if S ⊨ G then Terminate{solution}

    S' := Apply(advance-time,S)
    if ∃ ⟨p_i, t_i⟩∈G such that
        t_i < Time(S') and p_i ∉S then
        Terminate{non-solution}
    else S := S'
end while;
```

Figure 4: Algorithm to build the relaxed temporal planning graph structure.

S using the set of relaxed actions, which are generated from original actions by eliminating all effects which (1) delete some fact (predicate) or (2) reduce the level of some resource. Since delete effects are ignored, RTPG will not contain any mutex relations, which considerably reduces the cost of constructing RTPG. The algorithm to build the RTPG structure is summarized in Figure 4. To build the RTPG, we need three main datastructures: a fact level, an action level, and an unexecuted event queue.[6] Each fact *f* or action *A* is marked *in*, and appears in the RTPG's fact/action level at time instant t_f/t_A if it can be achieved/executed at t_f/t_A. In the beginning, only facts which appear in *P* are marked *in* at *t*, the action level is empty, and the event queue holds all the unexecuted events in *Q* that add new predicates. Action *A* will be marked *in* if (1) *A* is not already marked *in* and (2) all of *A*'s preconditions are marked *in*. When action *A* is *in*, then all of *A*'s unmarked instant add effects will also be marked *in* at *t*. Any delayed effect *e* of *A* that adds fact *f* is put into the event queue *Q* if (1) *f* is not marked *in* and (2) there is no event e' in *Q* that is scheduled to happen before *e* and which also adds *f*. Moreover, when an event *e* is added to *Q*, we will take out from *Q* any event e' which is scheduled to occur after *e* and also adds *f*.

When there are no more unmarked applicable actions in *S*, we will stop and return *no-solution* if either (1) *Q* is empty or (2) there exists some unmarked goal with a deadline that is smaller than the time of the earliest event in *Q*. If none of the situations above occurs, then we will apply *advance-time* action to *S* and activate all events at time point $t_{e'}$ of the earliest event *e'* in *Q*. The process above will be repeated until all the goals are marked *in* or one of the conditions indicating *non-solution* occurs. Figure 5 shows the RTPG for the state *S* at time point t_1 (refer to Figure 1) after we apply action *Board(P1)* to the initial state and *advance* the clock from t_0 to t_1.

In *Sapa*, the RTPG is used to:

- Prune the states that can not lead to any solution.
- Use the time points at which goals appear in the RTPG as the lower bounds on their time of achievements in the real plans.
- Build a relaxed plan that achieves the goals, which can then be used as a basis to estimate the distance from S to the goals.

For the first task, we will prune a state if there is some goal $\langle p_i, t_i \rangle$ such that p_i does not appear in the RTPG before time point t_i.

Proposition 1: *Pruning a state according to the relaxed temporal planning graph (RTPG) preserves the completeness of the planning algorithm.*

The proof is quite straight forward. Since we relaxed the delete effects and resource related constraints of all the actions when building the graph structure, and applied all applicable actions to each state, the time instant at which each predicate appears in the RTPG is a *lower bound* on its real time of achievement. Therefore, if we can not achieve some goal on time in the relaxed problem, then we definitely will not be able to achieve that goal with the full set of constraints.

In the next several sections, we will discuss the second task, that of deriving different heuristic functions from the RTPG structure.

3.2 Admissible heuristics based on action durations and deadlines

In this section, we will discuss how several admissible heuristic functions can be derived from the RTPG. First, from the observation that all predicates appear at the earliest possible time in the relaxed plan graph, we can derive an admissible heuristic which can be used to *optimize the makespan* of the solution. The heuristic is defined as follows:

Max-span heuristic: *Distance from a state to the goals is equal to the length of the duration between the time-instant of that state and the time the last goal appears in the RTPG .*

[6] Unlike the initial state, the event queue of the state *S* from which we build the RTPG may be.

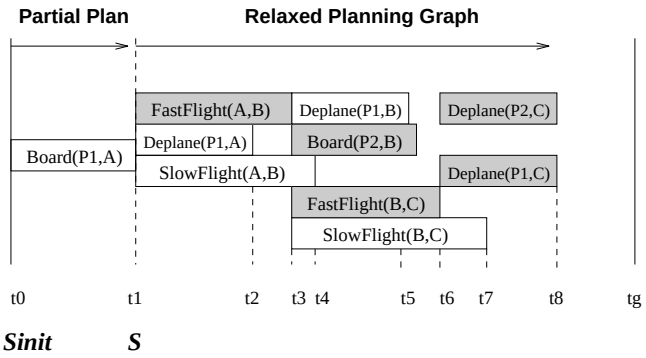


Figure 5: Sample relaxed temporal planning graph for durative actions. Shaded actions are the ones appear in the relaxed plan.

The max-span heuristic is admissible and can be used to find the smallest makespan solution for the planning problem. The proof of admissibility is based on the same observation made in the proof of Proposition 1. Because all the goals appear in the RTPG at the time instants that are *lower bounds* on the their real time of achievements, the time instant at which the last goal appears in the RTPG will be the lower bound on the actual time point at which we can achieve all the goals. Thus, it is a lower bound on the makespan of the solution.

The max-span heuristic discussed so far can be thought of as a generalized version of the max-action heuristic used in HSP [3] or max-level heuristic in AltAlt [12]. One of the assumptions in classical planning is that the goals have no deadlines and they need only be achieved by the end of the plan. Therefore, all heuristics concentrate on measuring how far the current state is to the point by which *all* the goals are achieved. However, in temporal planning with deadline goals, we can also measure the 'slack' values for the goals as another plan quality measurement (where slack is the difference in time between when the goal was achieved in the paln, and the deadline specified for its achievement). The slack values for a given set of goals can also be a good indication on how hard it is to achieve those goals, and thus, how hard it is to solve a planning problem from a given state. Moreover, slack-based objective functions are common in scheduling.

We will consider objective functions to maximize the minimum, maximum, or summation of slack values of all the goals for the temporal planning problems. In our case, the slack value for a given goal g is estimated from the RTPG by taking the difference between the time instant at which g appears in the RTPG and its deadline. We now present admissible heuristics for these three slack based objective functions.

Min-slack heuristic: *Distance from a state to the goals is equal to the minimum of the slack estimates of all individual goals.*[7]

Max-slack heuristic: *Distance from a state to the goals is equal to the maximum of slack estimates of all individual goals.*

Sum-slack heuristic: *Distance from a state to the goals is equal to the summation of slack estimates for all individual goals.*

The min-slack, max-slack, and sum-slack heuristics target the objective functions of maximizing the minimum slack, maximum slack, and the summation of all slack values. The admissibility of the three heuristics for the respective objective functions can be proven using the same argument we made for the max-span heuristic. Specifically, we use the observation that all goals appear in the RTPG at time instants earlier than the actual time instants at which they can be achieved, to prove that the slack estimated calculated using the RTPG for any goal will be the *upper bound* on its actual slack value for the non-relaxed problem.

3.3 Heuristics for efficient satisficing search

We now focus on efficiently finding reasonable quality plans. In the last section, we discussed several admissible heuristics which can be used to find optimal solution according to some objective functions. However, admissible heuristics such as max-span and slack-based heuristics are only concerned about the time points at which goals are achieved, and not the length of the eventual plan. In classical planning, heuristics that use an estimate on the length of the plan have been shown to be more effective in controlling search [7, 12]. To estimate the length of the solution, these planners typically use a valid plan extracted from the relaxed planning graph (the relaxation typically involves ignoring negative interactions). We can use a similar heuristic for temporal planning.

Sum-action heuristic: *Distance from a state to the goals is equal to the number of actions in the relaxed plan.*

The relaxed plan can be built backward from the goals in a manner nearly identical to the procedure used in Graphplan algorithm[2] in classical planning. We first start with the goals and add actions that support them to the

[7] If all the goals have the same deadlines, then maximizing the minimum slack is equal to minimizing the makespan of the plan and the two heuristic values (max-span and min-slack) can be used interchangably.

solution. If we add an action to the solution, then its preconditions are also added to the set of current goals. The search continues until we "reach" the initial state (i.e the goals are entailed by the initial state). In our continuing example, the shaded actions in Figure 5 are the ones that appear in the relaxed plan when we search backward.

Finally, since actions have different durations, the sum of the durations of actions in the relaxed plan is another way to measure the difficulty in achieving the goals.

Sum-duration heuristic: *Distance from a state to the goals is equal to the sum of durations of actions in the relaxed plan.*

If all actions have the same durations, then the sum of durations of all actions in the relaxed plan will be equivalent to taking the number of actions in the plan. Thus, in this case, sum-action and sum-duration will perform exactly the same. Neither of these heuristics are admissible; searches using the sum-action or sum-duration heuristics do not guarantee to return the solutions with smallest number of actions, or solutions with smallest summation of action durations. The reason is that these two heuristics have their values based on a *first* (relaxed) plan found. There is no guarantee that that first relaxed plan will be smaller than the smallest real (non-relaxed) plan in terms of number of actions, or summation of durations of actions in the plan.

3.4 Using metric resource constraints to adjust heuristic values

The heuristics discussed in the last two sections have used the knowledge about durations of actions and deadline goals but not about resource consumption. By ignoring the resource related effects when building the relaxed plan, we may miss counting actions whose only purpose is to give sufficient resource-related conditions to other actions.[8] Consequently, ignoring resource constraints may reduce the quality of heuristic estimate based on the relaxed plan. We are thus interested in adjusting the heuristic values discussed in the last two sections to account for the resource constraints.

In real-world problems, most actions consume resources, while there are special actions that increase the levels of resources. Since checking whether the level of a resource is sufficient for allowing the execution of an action is similar to checking the predicate preconditions, one obvious approach to adjust the relaxed plan would be to add actions that provide that resource-related condition to the relaxed plan. For reasons discussed below, it turns out to be too difficult to decide which actions should be added to the relaxed plan to satisfy the given resource conditions. First, actions that consume/produce the same metric-resource may overlap over the RTPG and thus make it hard to reason about the resource level at each time point. In such cases, the best we can do is to find the upper bound and lower bound values on the value of some specific resource. However, the bounds may not be very informative in reasoning about the exact value. Second, because we do not know the values of metric resources at each time point, it is difficult to reason as to whether or not an action needs another action to support its resource-related preconditions. For example, in Figure 5, when we add the action *fast-flying*(B, C) to the relaxed plan, we know that that action will need $fuel(airplane) > 400$ as its precondition. However, without the knowledge about the value (level) of $fuel(airplane)$ at that time point, we can hardly decide whether or not we need to add another action to achieve that precondition. If we reason that the fuel level at the initial state ($fuel(airplane) = 500$) is sufficient for that action to execute, then we already miss one unavoidable $refuel(airplane)$ action (because most of the fuel in the initial state has been used for the other flying action, *fast-flying*(A, B)). A final difficulty is that because of the continuous nature of the metric resources, it is harder to reason if an action *gives* a resource-related effect to another action and whether or not it is logically relevant to do so. For example, suppose that we need to fly an airplane from *cityA* to *cityB* and we need to refuel to do so. Action $refuel(airplane, cityC)$ gives the fuel that the airplane needs, but it is totally irrelevant to the plan. Adding that action to the relaxed plan (and its preconditions to the goal set) will lead to the addition of irrelevant actions, and thus reduce the quality of heuristic estimates it provides.

In view of the above complications, we introduce a new way of readjusting the relaxed plan to take into account the resource constraints as follows: we first preprocess the problem specifications and find for each resource *R* an action A_R that can increase the amount of *R* maximally. Let Δ_R be the amount by which A_R increases *R*, and let $Dur(A_R)$ be the duration of A_R. Let $Init(R)$ be the level of resource *R* at the state *S* for which we want to compute the relaxed plan, and $Con(R)$, $Pro(R)$ be the total consumption and production of *R* by all actions in the relaxed plan. If $Con(R) > Init(R) + Pro(R)$, we use the following formula to adjust the heuristic values of the sum-action and sum-duration according to the resource consumption.

Sum-action heuristic value h:

$$h \leftarrow h + \sum_R \left\lceil \frac{Con(R) - (Init(R) + Pro(R))}{\Delta_R} \right\rceil$$

Sum-duration heuristic value h:

$$h \leftarrow h + \sum_R \frac{Con(R) - (Init(R) + Pro(R))}{\Delta_R} * Dur(A_R)$$

[8] For example, if we want to drive a truck to some place and the fuel level is low, by totally ignoring the resource related conditions, we will not realize that we may need to *refuel* the truck before *driving* it.

	sum-act		sum-act adjusted		sum-dur		sum-dur adjusted	
prob	time (s)	node	time (s)	node	time (s)	node	time (s)	node
zeno1	0.272	14/48	0.317	14/48	0.35	20/67	0.229	9/29
zeno2	92.055	304/1951	61.66	188/1303	-	-	-	-
zeno3	23.407	200/996	38.225	250/1221	7.72	60/289	35.757	234/1079
zeno4	-	-	37.656	250/1221	7.76	60/289	35.752	234/1079
zeno5	83.122	575/3451	71.759	494/2506	-	-	-	-
zeno6	64.286	659/3787	27.449	271/1291	-	-	30.530	424/1375
zeno7	1.34	19/95	1.718	19/95	1.374	19/95	-	-
zeno8	1.11	27/87	1.383	27/87	1.163	27/87	1.06	14/60
zeno9	52.82	564/3033	16.310	151/793	130.554	4331/5971	263.911	7959/10266
log_p1	2.215	27/159	2.175	27/157	2.632	33/192	2.534	33/190
log_p2	165.350	199/1593	164.613	199/1592	37.063	61/505	-	-
log_p3	-	-	20.545	30/215	-	-	-	-
log_p4	13.631	21/144	12.837	21/133	-	-	-	-
log_p5	-	-	28.983	37/300	-	-	-	-
log_p6	-	-	37.300	47/366	-	-	-	-
log_p7	-	-	115.368	62/531	-	-	-	-
log_p8	-	-	470.356	76/788	-	-	-	-
log_p9	-	-	220.917	91/830	-	-	-	-

Table 2: Solution times and explored/generated search nodes for *Sapa* in the zeno-flying and temporal logistics domains with sum-action and sum-duration heuristics with/without resource adjustment technique. Times are in seconds. All experiments are run on a Sun Ultra 5 machine with 256MB RAM. "-" indicates that the problem can not be solved in 500 seconds.

We will call the newly adjusted heuristics **adjusted sum-action** and **adjusted sum-duration**. The basic idea is that even though we do not know if an individual resource-consuming action in the relaxed plan needs another action to support its resource-related preconditions, we can still adjust the number of actions in the relaxed plan by reasoning about the total resource consumption of *all* the actions in the plan. If we know how much excess amount of a resource *R* the relaxed plan consumes and what is the maximum increment of *R* that is allowed by any individual action in the domain, then we can infer the minimum number of resource-increasing actions that we need to add to the relaxed plan to balance the resource consumption.

For example, in the relaxed plan for our sample problem, we realize that the two actions *fast-flying*(A, B) and *fast-flying*(B, C) consume a total of: 1000/2 + 1200/2 = 1100 units of fuel, which is higher than the initial fuel level of 500 units. Moreover, we know that the maximum increment for the airplane's fuel is 750 for the *refuel(airplane)* action. Therefore, we can infer that we need to add at least $\lceil (1100 - 500)/750 \rceil = 1$ refueling action to make the relaxed plan consistent with the resource consumption constraints. The experimental results in Section 4 show that the metric resource related adjustments are quite important in domains which have many actions consuming different types of resources.

The adjustment approach described above is useful for improving the sum-action and sum-duration heuristics, but it can not be used for the max-span and slack-based heuristics without sacrificing their admissibility. In future, we intend to investigate the resource constraint-based adjustments for those heuristics that still preserve their admissibility.

4 Experimental results

We have implemented *Sapa* in Java. To date, our implementation of *Sapa* has been primarily used to test the performance of different heuristics and we have spent little effort on code optimization. We were primarily interested in seeing how effective the heuristics were in controlling the search. In the case of heuristics for satisficing search, we were also interested in evaluating the quality of the solution. We evaluate the performance of *Sapa* on problems from two metric temporal planning domains to see how well it performs in these complex planning problems. The first one is the zeno-flying domain discussed in Section 2.2 [14]. The second is our version of the temporal and metric resource version of the logistics domain. In this domain, trucks move packages between locations within one city, and planes carry them from one city to another. Different airplanes and trucks move with different speeds, have different fuel capacities, different fuel-consumption-rates, and different fuel-fill-rates when refueling. The temporal logistics domain is more complicated than the zeno-flying domain because it has more types of resource-consuming actions. Moreover, the *refuel* action in this domain has a dynamic duration, which is not the case for any action in the zeno-flying domain. Specifically, the duration of this action depends on the fuel level of the vehicle and can only be decided at the time we execute that action.

	sum-act		sum-act adjusted		sum-dur		sum-dur adjusted	
prob	#act	duration	#act	duration	#act	duration	#act	duration
zeno1	5	320	5	320	5	320	5	320
zeno2	23	1020	23	950	-	-	-	-
zeno3	22	890	13	430	13	450	17	400
zeno4	-	-	13	430	13	450	17	400
zeno5	20	640	20	590	-	-	-	-
zeno6	16	670	15	590	-	-	14	440
zeno7	10	370	10	370	10	370	-	-
zeno8	8	320	8	320	8	320	8	300
zeno9	14	560	13	590	13	460	13	430
log_p1	16	10.0	16	10.0	16	10.0	16	10.0
log_p2	22	18.875	22	18.875	22	18.875	-	-
log_p3	-	-	12	11.75	-	-	-	-
log_p4	12	7.125	12	7.125	-	-	-	-
log_p5	-	-	16	14.425	-	-	-	-
log_p6	-	-	21	18.55	-	-	-	-
log_p7	-	-	27	24.15	-	-	-	-
log_p8	-	-	27	19.9	-	-	-	-
log_p9	-	-	32	26.25	-	-	-	-

Table 3: Number of actions and duration (makespan) of the solutions generated by *Sapa* in the zeno-flying and logistics domains with sum-action and sum-duration heuristics with/without resource adjustment technique.

Table 2 and 3 summarize the results of our empirical studies. Before going into the details, we should mention that among the different types of heuristics discussed in the Section 3, max-span and slack-value based heuristics are admissible. However, they do not scale up to reasonable sized problems. As a matter of fact, the max-span heuristic can not solve any problems in Table 2 in the allotted time. The sum-slack heuristic returns an optimal solution (in terms of makespan and sum-slack values) for the problem *Zeno1* in zeno-flying domain in 7.3 seconds, but can not solve any other problems. However, both are able to solve smaller problems that are not listed in our result tables. Because of this, most of our remaining discussion is directed towards sum-action and sum-duration heuristics.

Table 2 shows the running times of *Sapa* for the *sum-action* and *sum-duration* heuristics with and without metric resource constraint adjustment technique (refer to Section 3.4) in the two planning domains discussed above. We tested with 9 problems from each domain. Most of the problems require plans of 10-30 actions, which are quite big compared to problems solved by previous domain-independent temporal planners reported in the literature. The results show that most of the problems are solved within a reasonable time (e.g under 500 seconds). More importantly, the number of nodes (time-stamped states) explored, which is the main criterion used to decide how well a heuristic does in guiding the search, is quite small compared to the size of the problems. In many cases, the number of nodes explored by the best heuristic is only about 2-3 times the size of the plan.

In general, the sum-action heuristic performs better than the sum-duration heuristic in terms of planning time, especially in the logistics domain. However, there are several problems in which the sum-duration heuristic returns better solving times and smaller number of nodes. The metric resource adjustment technique greatly helps the sum-action heuristic, especially in the logistics domain, where without it *Sapa* can hardly solve the bigger problems. We still do not have a clear answer as to why the resource-adjustment technique does not help the sum-duration heuristic.
Plan Quality: Table 3 shows the number of actions in the solution and the duration (makespan) of the solution for the two heuristics analyzed in Table 2. These categories can be seen as indicative of the problem's difficulty, and the quality of the solutions. By closely examining the solutions returned, we found that the solutions returned by *Sapa* have quite good quality in the sense that they rarely have many irrelevant actions. The absence of irrelevant actions is critical in the metric temporal planners as it will both save resource consumption and reduce execution time. It is interesting to note here that the temporal TLPlan[1], whose search algorithm *Sapa* adapts, usually outputs plans with many more irrelevant actions. Interestingly, Bacchus & Ady mention that their solutions are still better than the ones returned by LPSAT[19], which makes our solutions that much more impressive compared to LPSAT.

The pure sum-action heuristic without resource adjustment normally outputs plans with slightly higher number of actions, and longer makespans than the sum-duration heuristic. In some cases, the sum-action heuristic guides the search into paths that lead to very high makespan values, thus violating the deadline goals. After that, the planner has harder time getting back on the right track. Examples of this are zeno-4 and log-p3 which cannot be solved with sum-action heuristic if the deadlines are about 2 times smaller than the optimal makespan (because the search paths keep extending the time beyond the deadlines). The resource adjustment technique not only improves the sum-action heuristic in solution times, but also generally shortens the makespan and occasionally reduces the number of actions in the plan as well. As mentioned earlier, the adjustment technique generally does not help the sum-duration heuristics

in solving time, but it does help reduce the makespan of the solution in most of the cases where solutions can be found. However, the set of actions in the plan is generally still the same, which suggests that the adjustment technique does not change the solution, but *pushes* the actions up to an earlier part of the plan. Thus, it favors the execution of concurrent actions instead of using the special action *advance-time* to advance the clock.

When implementing the heuristics, one of the decisions we had to make was whether to recalculate the heuristic value when we advance the clock, or to use the same value as that of the parent node. On the surface, this problem looks trivial and the correct way seems to be to recalculate the heuristic values. However, in practice, keeping the parent node's heuristic value when we advance the clock always seems to lead to solutions with equal or slightly better makespan. We can explain the improved makespan by the fact that recalculating the heuristic value normally favors the *advance-clock* action by outputting a smaller heuristic value for it than the parent. Using many such *advance-clock* actions will lead to solutions with higher makespan values. The solving time comparison is somewhat mixed. Keeping the parent heuristics value speeds up 6 of the 9 problems tested in the logistics domain by average of 2x and slows down about 1.5x in the 3 zeno-flying problems. We do not have a clear answer for the solution time differences between the two approaches. In the current implementation of *Sapa*, we keep the parent node's heuristic value when we advance the clock.

Although we wanted to compare *Sapa* to other planners, there are very few implementations of metric temporal planners with capabilities comparable to *Sapa* that are publicly available and even they tend to scale up poorly. For example, although Zeno is a more expressive planner than *Sapa*, it can not scale up to bigger problems. The easiest problem in the zeno-flying domain in Table 2 (*Zeno1*) is reported in [14] to be solved by Zeno in several minutes with hand-coded domain control rules.[9] IxTeT is another known planner that we would have like to compare to, but the code is not available and IxTeT's results reported in the literature have concentrated on a class of temporal problems that use discrete, but not metric, resources. In the near future, we intend to compare our planner with TGP[16] and TP4[6] on a simpler set of temporal planning problems that can be handled by all three of them.

5 Related work

There have been several temporal planning systems in the literature that can handle different types of temporal and resource constraints. Among them, planners such as temporal TLPlan[1], Zeno[14], IxTeT[9], and HSTS[11] can solve problems that are similar to the one solved by *Sapa*. There are also planners such as Resource-IPP[8], TP4[6], TGP[16], and LPSAT[19] that can handle a subset of the types of problems discussed in this paper.

Closest to our work is the temporal TLPlan [1], which originates the algorithm to support concurrent actions in the forward state space search. The critical difference between this planner and *Sapa* is that while temporal TLPlan is controlled by hand-coded domain-specific control rules, *Sapa* uses domain-independent heuristics. Experimental results reported in [1] indicate that while Temporal TLPlan is very fast, but it tends to output plans with many irrelevant actions.

There are several partial order planners that can handle various types of temporal and resource constraints. Zeno[14] can solve problems with a wide range of constraints, as well as actions with conditional and quantified effects. However, Zeno lacks heuristic control and scales poorly. IxTeT[9] is another hierarchical partial order planner that can handle many types of temporal and resource constraints. Most of IxTeT's interesting innovations have been aimed at on handling discrete resources such as robots or machines but not on metric resources. HSTS[11] is a partial order planner that has been used to solve NASA temporal planning problems. Like TLPlan, HSTS uses hand-coded domain control knowledge to guide its search. parcPlan[10] is a domain-independent temporal planner using the least-commitment approach. parcPlan claims to be able to handle a rich set of temporal constraints, but the experiments in [10] do not demonstrate its expressiveness adequately.

Resource-IPP (RIPP)[8] is an extension of the IPP planner to deal with durative actions that may consume metric resources. RIPP considers time as another type of resource and solves the temporal planning problem by assuming that actions are still instantaneous. Like IPP, RIPP is based on Graphplan[2] algorithm. A limited empirical evaluation of RIPP is reported in [8]. TP4[6] by Haslumn & Geffner is a recent planner that employs backward chaining state space search with temporal or resource related admissible heuristics. The results of TP4 are promising in a subset of temporal planning problems where durations are measured in unit time, and resources decrease monotonically.

There are several planners in the literature that handle either temporal or resource constraints (but not both). TGP[16] is a temporal planner based on the Graphplan algorithm. TGP extends the notion of mutual exclusion relations in the Graphplan algorithm to allow constraints between actions and propositions. RTPG can be seen as a relaxed version of the planning graph that TGP uses. While TGP might provide better bounds on slacks and times of achievement, it is also costlier to compute. Cost of computation is especially critical as *Sapa* would have to compute the planning graph once for each expanded search node. It is nevertheless worth investigating the overall effectiveness of heuristics derived from TGP's temporal planning graph. LPSAT[19] can handle metric resource constraints by combining SAT and linear programming. As noted in Section 4, LPSAT seems to suffer from poor quality plans.

[9] We tried to run Zeno on the same machine used to test *Sapa* without control-knowledge for that problem, but Zeno indicated that it can not solve and returned a partial solution.

6 Conclusion and future work

In this paper, we described *Sapa*, a domain-independent forward chaining heuristic temporal planner that can handle metric resource constraints, actions with continuous duration, and deadline goals. *Sapa* does forward search in the space of time-stamped states. Our main focus has been on developing effective heuristics to control the search. We considered both satisficing and optimizing search scenarios and proposed effective heuristics for both. Our heuristics are based on the relaxed temporal planning graph structure. For optimizing search, we introduced admissible heuristics for objective functions based on the makespan and slack values. For satisficing search, we looked at heuristics such as sum-action and sum-duration, that are based on plans derived from RTPG. These were found to be quite efficient in terms of planning time. We also presented a novel technique to improve the heuristic values by reasoning about the metric resource constraints. Finally, we provided an extensive empirical evaluation demonstrating the performance of *Sapa* in several metric temporal planning domains.

In the near term, we intend to investigate the problem of finding better relaxed plans with regard to the resource and temporal constraints of actions in the domain. We are interested in how to use the resource time maps discussed in [8] in constructing the relaxed plan. Moreover, we want to use the binary mutex information, *a la* TGP [16] to improve heuristics in both optimizing and satisficing searches. Our longer term plans include incorporating *Sapa* in a loosely-coupled architecture to integrate planning and scheduling, which will be the logical continuation of our work with the *Realplan* system[18].

References

[1] Bacchus, F. and Ady, M. 2001. Planning with Resources and Concurrency: A Forward Chaining Approach. *Proc IJCAI-2001*.

[2] Blum, A. and Furst, M. 1995. Fast planning throught planning graph analysis. *Proc IJCAI-95*.

[3] Bonet, B., Loerincs, G., and Geffner, H. 1997. A robust and fast action selection mechanism for planning. *Proc AAAI-97*

[4] Fox, M. and Long, D. 2001. PDDL+: An Extension to PDDL for Expressing Temporal Domains

[5] Ghallab, M. and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. *Proc AIPS-94*

[6] Haslum, P. and Geffner, H. 2001. Heuristic Planning with Time and Resources *Workshop on Planing with Rersource, IJCAI-01*

[7] Hoffmann, J. 2000. http://www.informatik.uni-freiburg.de/ hoffmann/ff.html

[8] Koehler, J. 1998. Planning under Resource Constraints. *Proc ECAI-98*

[9] Laborie, P. and Ghallab, M. 1995. Planning with sharable resource constraints. *Proc IJCAI-95*.

[10] Liatsos, V., and Richards, B. 1999. Scaleability in Planning. *Proc ECP-99*.

[11] Muscettola, N. 1994. Integrating planning and scheduling. *Intelligent Scheduling.*

[12] Nguyen, X., Kambhampati, S., and Nigenda, R. 2001. Planning Graph as the Basis for deriving Heuristics for Plan Synthesis by State Space and CSP Search. *To appear in Artificial Intelligence.*

[13] Pearl, J. 1985. Heuristics. *Addison-Wesley*

[14] Penberthy, S. and Well, D. 1994. Temporal Planning with Continuous Change. *Proc AAAI-94*.

[15] Pinedo, M. 1995. Scheduling: Theory, Algorithms, and Systems. *Prentice Hall*

[16] Smith, D. and Weld, D. 1999. Temporal Planning with Mutual Exclusion Reasoning. *Proc IJCAI-99*

[17] Smith, D., Frank J., and Jonsson A. 2000. Bridging the gap between planning and scheduling. *The Knowledge Engineering Review*, Vol. 15:1.

[18] Srivastava, B., Kambhampati, S., and Do, M. 2001. Planning the Project Management Way: Efficient Planning by Effective Integration of Causal and Resource Reasoning in RealPlan. *To appear in Artificial Intelligence.*

[19] Wolfman, S. and Weld, D. 1999. Combining Linear Programming and Satisfiability Solving for Resource Planning. *Proc IJCAI-99*

Heuristic Planning with Time and Resources

Patrik Haslum[1] and Héctor Geffner[2]

[1] Department of Computer Science, Linköping University, Sweden
pahas@ida.liu.se
[2] Departamento de Computación, Universidad Simón Bolívar, Venezuela
hector@usb.ve

Abstract We present an algorithm for planning with time and resources, based on heuristic search. The algorithm minimizes makespan using an admissible heuristic derived automatically from the problem instance. Estimators for resource consumption are derived in the same way. The goals are twofold: to show the flexibility of the heuristic search approach to planning and to develop a planner that combines expressivity and performance. Two main issues are the definition of *regression* in a temporal setting and the definition of the *heuristic* estimating completion time. A number of experiments are presented for assessing the performance of the resulting planner.

1 Introduction

Recently, heuristic state space search has been shown to be a good framework for developing different kinds of planning algorithms. It has been most successful in non-optimal sequential planning, *e.g.* HSP [4] and FF [10], but has been applied also to optimal and parallel planning with good results [8].

We continue this thread of research by developing a domain-independent planning algorithm for domains with metric time and certain kinds of resources. The algorithm relies on regression search guided by a heuristic that estimates completion time and which is derived automatically from the problem representation. The algorithm minimizes the overall execution time of the plan, commonly known as the *makespan*.

As far as we are aware, no effective domain-independent planner matches the expressivity of our planner, though some exhibit common features. For example, TGP [22] handles actions with duration and optimizes makespan, while RIPP [13] and GRT-R [20] handle resources, and are in this respect more expressive than our planner.

Among planners that exceed our planner in expressivity, e.g. Zeno [18], IxTeT [7] and HSTS [17], none have reported significant domain-independent performance (Jonsson *et al.* [11] describe the need for sophisticated engineering of domain dependent search control for the HSTS planner). Many highly expressive planners, *e.g.* O-Plan [24], ASPEN [5] or TALplanner [15],

are "knowledge intensive", relying on user-provided problem decompositions, evaluation functions or search constraints[1].

2 Action Model and Assumptions

The action model we use is propositional STRIPS with extensions for time and resources. As in GRAPHPLAN [3] and many other planners, the action set is enriched with a *no-op* for each atom p which has p as its only precondition and effect. Apart from having a variable duration, a no-op is viewed and treated like a regular action.

2.1 Time

When planning with time each action a has a duration, $dur(a) > 0$. We take the time domain to be $\mathbb{R}^+$. In most planning domains we could use the positive integers, but we have chosen the reals to highlight the fact that the algorithm does not depend on the existence of a least indivisible time unit. Like Smith and Weld [22], we make the following assumptions: For an action a executed over an interval $[t, t + dur(a)]$

(i) the preconditions $pre(a)$ must hold at t, and preconditions not deleted by a must hold throughout $[t, t + dur(a)]$ and
(ii) the effects $add(a)$ and $del(a)$ take place at some point in the interior of the interval and can be used only at the end point $t + dur(a)$.

Two actions, a and b, are *compatible* iff they can be safely executed in overlapping time intervals. The above assumptions lead to the following condition for compatibility: a and b are compatible iff for each atom $p \in pre(a) \cup add(a)$, $p \notin del(b)$ and vice versa (*i.e.* $p \in pre(b) \cup add(b)$ implies $p \notin del(a)$).

2.2 Resources

The planner handles two types of resources: *renewable* and *consumable*. Renewable resources are needed during the execution of an action but are not consumed (*e.g.* a machine). Consumable resources, on the other hand, are consumed or produced (*e.g.* fuel). All resources are treated as real valued quantities; the division into unary, discrete and continuous is determined by

[1] The distinction is sometimes hard to make. For instance, *parc*Plan [16] domain definitions appear to differ from plain STRIPS only in that negative effects of actions are modeled indirectly, by providing a set of constraints, instead of explicitly as "deletes". *parc*Plan has shown good performance in certain resource constrained domains, but domain definitions are not available for comparison.

the way the resource is used. Formally, a planning problem is extended with sets R_P and C_P of renewable and consumable resource names. For each resource name $r \in R_P \cup C_P$, $avail(r)$ is the amount initially available and for each action a, $use(a, r)$ is the amount used or consumed by a.

3 Planning with Time

We describe first the algorithm for planning with time, not considering resources. In this case, a *plan* is a set of action instances with starting times such that no incompatible actions overlap in time, action preconditions hold over the required intervals and goals are achieved on completion. The cost of a plan is the total execution time, or *makespan*. We describe each component of the search scheme: the search space, the branching rule, the heuristic, and the search algorithm.

3.1 Search Space

Regression in the classical setting is a search in the space of "plan tails", *i.e.* partial plans that achieve the goals provided that the preconditions of the partial plans are met. A regression state, *i.e.* a set of atoms, *summarizes* the plan tail; if s is the state obtained by regressing the goal through the plan tail P' and P is a plan that achieves s from the initial state, then the concatenation of P and P' is a valid plan. A similar decomposition is exploited in the forward search for plans.

In a temporal setting, a set of atoms is no longer sufficient to summarize a plan tail or plan head. For example, the set s of all atoms made true by a plan head P at time t holds no information about the actions in P that have started but not finished before t. Then, if a plan tail P' maps s to a goal state, the combination of P and P' is not necessarily a valid plan. To make the decomposition valid, search states have to be extended with the actions under execution and their completion times. Thus, in a temporal setting states become pairs $s = (E, F)$, where E is a set of atoms and $F = \{(a_1, \delta_1), \ldots, (a_n, \delta_n)\}$ is a set of actions a_i with time increments δ_i.

An alternative representation for plans will be useful: instead of a set of time-stamped action instances, a plan is represented by a sequence $\langle (A_0, \delta_0), \ldots, (A_m, \delta_m) \rangle$ of action sets A_i and positive time increments δ_i. Actions in A_0 begin executing at time $t_0 = 0$ and actions in A_i, $i = 1 \ldots m$, at time $t_i = \sum_{0 \leqslant j < i} \delta_j$ (*i.e.* δ_i is the time to wait between the beginning of actions A_i and the beginning of actions A_{i+1}).

State Representation A search state $s = (E, F)$ is a pair consisting of a set of atoms E and a set of actions with corresponding time increments $F = \{(a_1, \delta_1), \ldots, (a_n, \delta_n)\}$, $0 < \delta_i \leq dur(a_i)$. A plan P *achieves* $s = (E, F)$ at time t if P makes all the atoms in E true at t and schedules the actions a_i at time $t - \delta_i$. The initial search state is $s_0 = (G_P, \emptyset)$, where G_P is the goal set of the planning problem. Final states are all $s = (E, \emptyset)$ such that $E \subseteq I_P$.

Branching Rule A successor to a state $s = (E, F)$ is constructed by selecting for each atom $p \in E$ an establisher (*i.e.* a regular action or no-op a with $p \in add(a)$), subject to the constraints that the selected actions are compatible with each other and with each action $b \in F$, and that at least one selected action is not a no-op. Let SE be the set of selected establishers and let

$$F_{\text{new}} = \{(a, dur(a)) \mid a \in SE\}.$$

The new state $s' = (E', F')$ is defined as the atoms E' that must be true and the actions F' that must be executing before the last action in $F \cup F_{\text{new}}$ begins. This will happen in a time increment δ_{adv}:

$$\delta_{adv} = \min\{\delta \mid (a, \delta) \in F \cup F_{\text{new}} \text{ and } a \text{ is not a } \textit{no-op}\}$$

where no-op actions are excluded from consideration since they have variable duration (the meaning of the action no-op(p) in s is that p has persisted in the *last* time slice). Setting the duration of no-ops in F_{new} equal to δ_{adv}, the state $s' = (E', F')$ that succeeds $s = (E, F)$ becomes

$$E' = \{pre(a) \mid (a, \delta_{adv}) \in F \cup F_{\text{new}}\}$$
$$F' = \{(a, \delta - \delta_{adv}) \mid (a, \delta) \in F \cup F_{\text{new}}, \delta > \delta_{adv}\}$$

The cost of the transition from s to s' is $c(s, s') = \delta_{adv}$ and the fragment of the plan tail that corresponds to the transition is

$$P(s, s') = (A, \delta_{adv}), \text{ where } A = \{a \mid (a, \delta_{adv}) \in F \cup F_{new}\}$$

The accumulated cost (plan tail) along a state-path is obtained by adding up (concatenating) the transition costs (plan fragments) along the path. The accumulated cost of a state is the minimum cost along all the paths leading to s. The evaluation function used in the search algorithm adds up this cost and the heuristic cost defined below.

Properties The branching rule is sound in the sense that it generates only valid plans, but it does not generate *all* valid plans. This is actually a desirable feature[2]. The rule is *optimality preserving* in the sense that it generates *some* optimal plan. This, along with soundness, is all that is needed for optimality (provided an admissible search algorithm and heuristic are used).

3.2 Heuristic

As in previous work [8], we derive an admissible heuristic by introducing approximations in the recursive formulation of the optimal cost function.

For any state $s = (E, F)$, the optimal cost is $H^*(s) = t$ iff t is the least time t such that there is a plan P that achieves s at t. The optimal cost function, H^*, is the solution to the Bellman equation [2]:

$$H^*(s) = \begin{cases} 0 & \text{if } s \text{ is final} \\ \min_{s' \in R(s)} c(s, s') + H^*(s') \end{cases} \quad (1)$$

where $R(s)$ is the regression set of s, *i.e.* the set of states that can be constructed from s by the branching rule.

Approximations. Since equation (1) cannot be solved in practice, we derive a lower bound on H^* by considering some inequalities. First, since a plan that achieves the state $s = (E, F)$, for $F = \{(a_i, \delta_i)\}$, at time t must achieve the preconditions of the actions a_i at time $t - \delta_i$ and these must remain true until t, we have

$$H^*(E, F) \geqslant \max_{(a_k, \delta_k) \in F} H^*(\bigcup_{(a_i, \delta_i) \in F, \delta_i \geqslant \delta_k} pre(a_i), \emptyset) + \delta_k \quad (2)$$

$$H^*(E, F) \geqslant H^*(E \cup \bigcup_{(a_i, \delta_i) \in F} pre(a_i), \emptyset) \quad (3)$$

Second, since achieving a set of atoms E implies achieving each subset E' of E we also have

$$H^*(E, \emptyset) \geqslant \max_{E' \subseteq E, |E'| \leqslant m} H^*(E', \emptyset) \quad (4)$$

where m is any positive integer.

[2] The plans generated are such that a regular action is executing during any given time interval and no-ops begin only at the times that some regular action starts. This is due to the way the temporal increments δ_{adv} are defined. Completeness could be achieved by working on the rational time line and setting δ_{adv} to the *gcd* of all actions durations, but as mentioned above this is not needed for optimality.

Temporal Heuristic H_T^m. We define a lower bound H_T^m on the optimal function H^* by transforming the above inequalities into equalities. A family of admissible temporal heuristics H_T^m for arbitrary $m = 1, 2, \ldots$ is then defined by the equations

$$H_T^m(E, \emptyset) = 0 \text{ if } E \subseteq I_P \tag{5}$$

$$H_T^m(E, \emptyset) = \min_{s' \in R(s=(E,\emptyset))} c(s = (E, \emptyset), s') + H_T^m(s') \text{ if } |E| \leqslant m \tag{6}$$

$$H_T^m(E, \emptyset) = \max_{E' \subseteq E, |E'| \leqslant m} H_T^m(E', \emptyset) \text{ if } |E| > m \tag{7}$$

$$H_T^m(E, F) = \max[\max_{(a_k, \delta_k) \in F} H_T^m(\bigcup_{(a_i, \delta_i) \in F, \delta_i \geqslant \delta_k} pre(a_i), \emptyset) + \delta_k \ , \\ H_T^m(E \cup \bigcup_{(a_i, \delta_i) \in F} pre(a_i), \emptyset)] \tag{8}$$

The relaxation is a result of the last two equations; the first two are also satisfied by the optimal cost function. Unfolding the right-hand side of equation (6) using (8), the first two equations define the function $H_T^m(E, F)$ completely for $F = \emptyset$ and $|E| \leq m$. From an implementation point of view, this means that for a fixed m, $H_T^m(E, \emptyset)$ can be solved and precomputed for all sets of atoms with $|E| \leq m$, and equations (7) and (8) used at run time to compute the heuristic value of arbitrary states.The precomputation is a simple variation of a shortest-path problem and its complexity is a low order polynomial in $|A|^m$, where $|A|$ is the number of atoms.

For a fixed m, equation (6) can be simplified because only a limited set of states can appear in the regression set. For example, for $m = 1$, the state s in (6) must have the form $s = (\{p\}, \emptyset)$ and the regression set $R(s)$ contain only states $s' = (pre(a), \emptyset)$ for actions a such that $p \in add(a)$. As a result, for $m = 1$, (6) becomes

$$H_T^1(\{p\}, \emptyset) = \min_{a: p \in add(a)} dur(a) + H_T^1(pre(a), \emptyset) \tag{9}$$

The corresponding equations for H_T^2 are in [9].

3.3 Search Algorithm

Any admissible search algorithm, *e.g.* A*, IDA* or DFS branch-and-bound [14], can be used with the search scheme described above to find optimal solutions.

The planner uses IDA* with some standard enhancements (cycle checking and a transposition table) and an optimality preserving pruning rule explained below. The heuristic used is H_T^2, precomputed for sets of at most two atoms as described above.

Incremental Branching In the implementation of the branching scheme, the establishers in SE are not selected all at once. Instead, this set is constructed incrementally, one action at a time. After each action is added to the set, the cost of the resulting "partial" state is estimated so that dead-ends (states whose cost exceeds the bound) are detected early. A similar idea is used in GRAPHPLAN. In a temporal setting, things are a bit more complicated because no-ops have a duration (δ_{adv}) that is not fixed until the set of establishers is complete. Still, a lower bound on this duration can be derived from the regular actions selected so far and in the state being regressed.

Selecting the Atom to Regress The order in which atoms are regressed makes no difference for completeness, but does affect the size of the resulting search tree. We regress the atoms in order of decreasing "difficulty": the difficulty of an atom p is given by the estimate $H^2_T(\{p\}, \emptyset)$.

Right-Shift Pruning Rule In a temporal plan there are almost always some actions that can be shifted forward or backward in time without changing the plan's structure or makespan (*i.e.* there is some "slack"). A right-shifted plan is one in which such movable actions are scheduled as late as possible.

As mentioned above, it is not necessary to consider all valid plans in order to guarantee optimality. In the implemented planner, non-right-shifted plans are excluded by the following rule: If s' is a successor to $s = (E, F)$, an action a compatible with all actions in F may *not* be used to establish an atom in s' when all the atoms in E' that a adds have been obtained from s by no-ops. The reason is that a could have been used to support the same atoms in E, and thus could have been shifted to the right (delayed).

4 Planning with Resources

Next, we show how the planning algorithm deals with renewable and consumable resources.

4.1 Renewable Resources

Renewable resources limit the set of actions that can be executed concurrently and therefore need to enter the planning algorithm only in the branching rule. When regressing a state $s = (E, F)$, we must have that

$$\sum_{(a_i,\delta_i)\in F\cup F_{\text{new}}} use(a_i, r) \leqslant avail(r) \tag{10}$$

for every renewable resource $r \in R_P$.

Heuristic The H_T^m heuristics remain admissible in the presence of renewable resources, but in order to get better lower bounds we exclude from the regression set any set of actions that violates a resource constraint. For unary resources (capacity 1) the resulting heuristic is informative, but for multi-capacity resources it tends to be weak.

4.2 Consumable Resources

To ensure that resources are not over-consumed, a state s must contain the remaining amount of each consumable resource r. For the initial state, this is $rem(s_0, r) = avail(r)$, and for a state s' resulting from s

$$rem(s', r) = rem(s, r) - \sum_{(a_i, t_i) \in F_{\text{new}}} use(a_i, r) \tag{11}$$

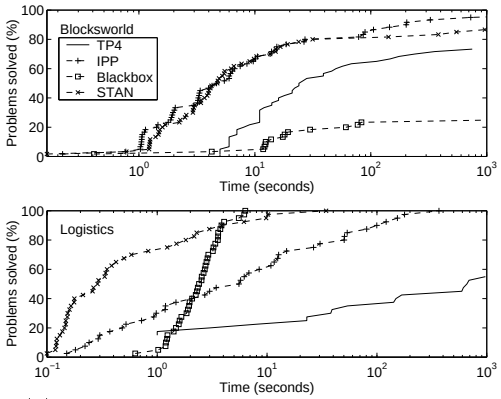
for each $r \in C_P$.

Heuristic The heuristics H_T^m remain admissible in the presence of consumable resources, but become less useful since they predict completion time but not conflicts due to overconsumption. If, however, consumable resources are restricted to be *monotonically decreasing* (*i.e.* consumed but not produced), a state s can be pruned if the amount of any resource r needed to achieve s from the initial situation exceeds the amount remaining in s, $rem(s, r)$. The amount needed is estimated by a function $need^m(s, r)$ defined in a way analogous to the function $H_T^m(s)$ that estimates time. The planner implements $need^1(s, r)$.

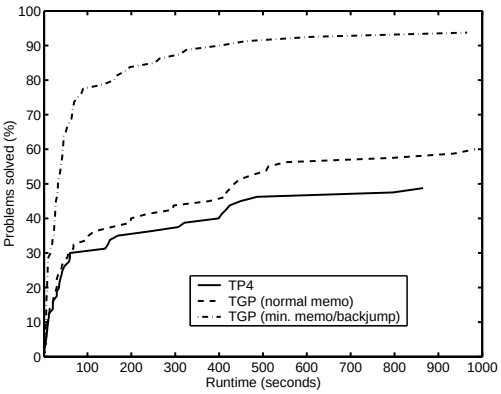
Because resource consumption is treated separately from time, this solution is weak when the time and resources needed to achieve a goal interact in complex ways. The H_T^m estimator considers only the fastest way of achieving the goal regardless of resource cost, while the $need^m$ estimator considers the cheapest way to achieve the goal regardless of time (and other resources). To overcome this problem, the estimates of time and resources would have to be integrated, as in for example [20]. Integrated estimates could also be used to optimize some combination of time and resources, as opposed to time alone.

4.3 Maintenance Actions

In planning, it is normally assumed that no explicit action is needed to maintain the truth of a fact once it has been established, but in many cases this assumption is not true. We refer to no-ops that consume resources as *maintenance actions.* Incorporating maintenance actions in the branching scheme outlined above is straightforward: For each atom p and each resource r, a



(a) Runtime distributions for TP4 and optimal non-temporal parallel planners on standard planning problems. A point $\langle x, y \rangle$ on the curve indicates that y percent of the problems were solved in x seconds or less. Note that the time axis is logarithmic.



(b) Runtime distributions for TP4 and TGP on problems from the simple temporal logistics domain.

Figure1.

quantity $use(\texttt{maintain}(p), r)$ can be provided as part of the domain definition and is set to 0 by default. Since the duration of a no-op is variable, we interpret $use(\texttt{maintain}(p), r)$ as the *rate* of consumption. For the rest, maintenance actions are treated as regular actions, and no other changes are needed in the planning algorithm.[3]

5 Experimental Results

We have implemented the algorithm for planning with time and resources described above, including maintenance actions but with the restriction that consumable resources are monotonically decreasing, in a planner called TP4[4]. The planner uses IDA* with some standard enhancements and the H^2_T heuristic. The resource consumption estimators consider only single atoms.

[3] This treatment of maintenance actions is not completely general. Recall that the branching rule does not generate *all* valid plans: in the presence of maintenance actions it may happen that some of the plans that are not generated demand less resources than the plans that are. When this happens, the algorithm may produce non-optimal plans or even fail to find a plan when one exists. This is a subtle issue that we will address in the future.

[4] TP4 is implemented in C. Planner, problems, problem generators and experiment scripts are available at `http://www.ida.liu.se/~pahas/hsps/`. Experiments were run on a Sun Ultra 10.

5.1 Non-Temporal Planning

First, we compare TP4 to three optimal parallel planners, IPP, BLACKBOX and STAN, on standard planning problems without time or resources. The test set comprises 60 random problems from the 3-operator blocksworld domain, ranging in size from 10 to 12 blocks, and 40 random logistics problems with $5--6$ deliveries. Blocksworld problems were generated using Slaney & Thiebaux's BWSTATES program [21].

Figure 1(a) presents the results in the form of runtime distributions. Clearly TP4 is not competitive with non-temporal planners, which is expected considering the overhead involved in handling time. Performance in the logistics domain, however, is very poor (*e.g.* TP4 solves less than 60% of the problems within 1000 seconds, while all other planners solve 90% within only 100 seconds), indicating that other causes are involved (most likely the branching rule, see below).

5.2 Temporal Planning

To test TP4 in a temporal planning domain, we make a small extension to the logistics domain[5], in which trucks are allowed to drive between cities as well as within and actions are assigned durations as follows:

Actions	Duration	Actions	Duration
Load/Unload	1	Drive truck (between cities)	12
Drive truck (within city)	2	Fly airplane	3

This is a simple example of a domain where makespan-minimal plans tend to be different from minimal-step parallel plans.

For comparison, we tested also the TGP planner [22]. The test set comprised 80 random problems with $4--5$ deliveries. Results are in figure 1(b). We tested two versions of TGP, one using plain GRAPHPLAN-like memoization and the other minimal conflict set memoization and "intelligent backtracking" [12]. TP4 shows a behaviour similar to the plain version of TGP, though somewhat slower. As the top curve shows, the intelligent backtracking mechanism is very effective in this domain (this was indicated also in [12]).

5.3 Planning with Time and Resources

Finally, for a test domain involving both time and non-trivial resource constraints we have used a scheduling problem, called multi-mode resource constrained project scheduling (MRCPS) [23]. The problem is to schedule a set

[5] The goal in the logistics domain is to transport a number of packages between locations in different cities. Trucks are used for transports within a city and airplanes for transports between cities. The standard domain is available *e.g.* as part of the AIPS 2000 Competition set [1].

of tasks and to select for each task a mode of execution so as to minimize project makespan, subject to precedence constraints among the tasks and global resource constraints. For each task, each mode has different duration and resource requirements. Resources include renewable and (monotonically decreasing) consumable. Typically, modes represent different trade-offs between time and resource use, or between use of different resources. This makes finding optimal schedules very hard, even though the planning aspect of the problem is quite simple.

The test comprised sets of problems with 12, 14 and 16 tasks and approximately 550 instances in each (sets `J12`, `J14` and `J16` from [19]). A specialized scheduling algorithm solves all problems in the set, the hardest in just below 300 seconds [23]. TP4 solves 59%, 41% and 31%, respectively, within the same time limit.

6 Conclusions

We have developed an optimal, heuristic search planner that handles concurrent actions, time and resources, and minimizes makespan. The two main issues we have addressed are the formulation of an admissible heuristic estimating completion time and a branching scheme for actions with durations. In addition, the planner incorporates an admissible estimator for consumable resources that allows more of the search space to be avoided. Similar ideas can be used to optimize a combination of time and resources as opposed to time alone.

The planner achieves a tradeoff between performance and expressivity. While it is not competitive with either the best parallel planners or specialized schedulers, it accommodates problems that do not fit into either class. An approach for improving performance that we plan to explore in the future is the combination of the lower bounds provided by the admissible heuristics H^m_T with a different branching scheme. See [6] for details.

Acknowledgments

We'd like to thank David Smith for much help with TGP. This research has been supported by the Wallenberg Foundation and the ECSEL/ENSYM graduate study program.

References

1. AIPS competition, 2000. http://www.cs.toronto.edu/aips2000/.
2. R.E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
3. A.L. Blum and M.L. Furst. Fast planning through graph analysis. *Artificial Intelligence*, 90(1–2):281 – 300, 1997.
4. B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 2001. To appear.
5. A.S. Fukunaga, G. Rabideau, S. Chien, and D. Yan. ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *Proc. International Symposium on AI, Robotics and Automation in Space*, 1997.
6. H. Geffner. Planning as branch and bound and its relation to constraint-based approches. http://www.ldc.usb.ve/∼hector.
7. M. Ghallab and H. Laruelle. Representation and control in IxTeT, a temporal planner. In *Proc. 2nd International Conference on AI Planning Systems*, 1994.
8. P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling*. AAAI Press, 2000.
9. P. Haslum and H. Geffner. Heuristic planning with time and resources. In *Proc. IJCAI Workshop on Planning with Resources*, 2001. http://www.ida.liu.se/∼pahas/hsps/.
10. J. Hoffman. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proc. 12th International Symposium on Methodologies for Intelligent Systems*, 2000.
11. A. Jonsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith. Planning in interplanetary space: Theory and practice. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 2000.
12. S. Kambhampati. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of AI Research*, 12:1 – 34, 2000.
13. J. Koehler. Planning under resource constraints. In *Proc. 13th European Conference on Artificial Intelligence*, 1998.
14. R.E. Korf. Artificial intelligence search algorithms. In *Handbook of Algorithms and Theory of Computation*, chapter 36. CRC Press, 1999.
15. J. Kvarnstrom and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1):119 – 169, 2000.
16. J.M. Lever and B. Richards. *parc*PLAN: A planning architecture with parallel actions, resources and constraints. In *Proc. 9th International Symposium on Methodologies for Intelligent Systems*, 1994.
17. N. Muscettola. Integrating planning and scheduling. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*. Morgan-Kaufmann, 1994.
18. J.S. Penberthy and D.S. Weld. Temporal planning with continous change. In *Proc. 12th National Conference on Artificial Intelligence (AAAI'94)*, 1994.
19. PSPLib: The project scheduling problem library. http://www.bwl.uni-kiel.de/Prod/psplib/library.html.
20. I. Refanidis and I. Vlahavas. Heuristic planning with resources. In *Proc. 14th European Conference on Artificial Intelligence*, 2000.
21. J. Slaney and S. Theibaux. Blocks world revisited. *Artificial Intelligence*, 125, 2001. See http://arp.anu.edu.au:80/∼jks/bw.html.
22. D.E. Smith and D.S. Weld. Temporal planning with mutual exclusion reasoning. In *Proc. 16th International Joint Conference on Artificial Intelligence*, 1999.
23. A. Sprecher and A. Drexl. Solving multi-mode resource-constrained project scheduling problems by a simple, general and powerful sequencing algorithm. I: Theory & II: Computation. Technical Report 385 & 386, Instituten für Betriebswirtschaftsrlehre der Universität Kiel, 1996.
24. A. Tate, B. Drabble, and J. Dalton. O-Plan: a knowledge-based planner and its application to logistics. In *Advanced Planning Technology*. AAAI Press, 1996.

Integrating Planning and Scheduling through Adaptation of Resource Intensity Estimates

Karen L. Myers[1], Stephen F. Smith[2], David W. Hildum[2], Peter A. Jarvis[1], Raymond de Lacaze[1]

[1] AI Center, SRI International, Menlo Park, CA USA
{myers,jarvis,delacaze}@ai.sri.com

[2] The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA USA
{sfs,hildum}@cs.cmu.edu

Abstract. We describe an incremental and adaptive approach to integrating hierarchical task network planning and constraint-based scheduling. The approach is grounded in the concept of approximating the 'resource intensity' of planning options. A given planning problem is decomposed into a sequence of (not necessarily independent) subtasks, which are planned and then scheduled in turn. During planning, operators are rated according to a heuristic estimate of their expected resource requirements. Options are selected that best match a computed 'target intensity' for planning. Feedback from the scheduler is used to adapt the target intensity after completion of each subplan, thus guiding the planner toward solutions that are tuned to resource availability. Experimental results from an air operations domain validate the effectiveness of the approach relative to typical "waterfall" models of planner/scheduler integration.

1. Introduction

Goal-oriented activity in complex domains typically requires a combination of planning and scheduling. A manufacturing facility must develop process plans for ordered parts that can be cost-effectively integrated with current production operations. Military planners must select courses of actions that achieve strategic objectives, while making the most of available assets. Space observatories must allocate viewing instruments to maximize scientific return under a large and diverse set of causal restrictions and dependencies. Though conceptually decomposable, planning and scheduling processes in such domains can be and often are highly interdependent. Different planning options for achieving a given objective can make quite different demands on system resources; correspondingly, current resource commitments and availability will impact the feasibility or desirability of various planning options.

The effectiveness of goal-oriented activity is ultimately tied to an ability to keep pace with evolving circumstances, and one recognized obstacle in practice is poor integration of "planning" and "scheduling" processes. In manufacturing organizations, this problem has been characterized as the "wall between engineering and manufacturing". Similar sorts of barriers can be found in other large-scale enterprises. The crux of the problem is lack of communication. Plans are developed with no visibility of resource availability and operational status, and likewise, schedules are developed and managed without knowledge of objectives and dependencies. Without such informa-

tion exchange, planning and scheduling processes are forced to each proceed in an uninformed and inherently inefficient manner. In the simplest case, the result is an *iterative waterfall* model of integration, where planning and scheduling are performed in sequential lockstep fashion and any problem encountered during scheduling simply triggers the generation of a new plan.

In this paper, we present a method for improving the overall planning and scheduling process through a tighter integration of these constituent activities. By planning, we refer generally to the process of deciding *what* to do; i.e., the process of transforming strategic objectives into executable activity networks. We use the term scheduling to refer alternatively to the process of deciding *when* and *how*; i.e., which resources to use to execute various activities and over what time frames. Traditionally, AI research has viewed planning and scheduling as distinct activities, and different solution techniques and technologies have emerged for each. Relatively few attempts have been made to combine respective technologies into larger integrated frameworks.

We take as our starting point previously developed technologies for hierarchical task network (HTN) planning and constraint-based scheduling. We describe and evaluate an approach to their integration based on the idea of approximating the resource requirements (called *resource intensity*) of different planning options, and incrementally exchanging and exploiting information about likely resource shortfalls and excesses to settle on options that best utilize available resources. Finally, we present experimental results that compare an implementation of the method to an iterative waterfall model of integration within the air operations domain. These results show that the intensity-based approach provides plans of comparable quality for greatly reduced computation time.

2. Technology Foundations

Planning The CPEF system provides the planning component for our work [8]. CPEF embodies a philosophy of plans as dynamic, open-ended artifacts that evolve in response to a continuously changing environment. CPEF provides a range of operations required for continuous plan management, including *plan generation*, *plan execution*, *monitoring*, and *plan repair*. Plan generation within CPEF is based on the CHIP system – an HTN planner derived from SIPE-2 [15].

Scheduling ACS, a constraint-based scheduler, provides the base scheduling capability. ACS is an air operations scheduler constructed using OZONE [13], a customizable constraint-based modeling and search framework for developing incremental scheduling tools. OZONE consolidates the results of application development experiences in a range of complex domains, including one recently deployed system for day-to-day management of airlift resources at the USAF Air Mobility Command (AMC) [1]. The ACS scheduler adapts techniques underlying the AMC application to the air operations domain. ACS can be used to generate, incrementally extend and revise assignments of aircraft and munitions to input target demands over time, taking into account priorities, desired levels of damage, time-on-target (TOT) windows, temporal sequencing constraints, feasible resource alternatives, and aircraft/munitions positioning and availability constraints.

3. Air Operations Domain Characteristics and Model

Applications that require integrated planning and scheduling will have individual characteristics that dictate the relative importance of each of these capabilities. Much of the work to date on combining AI planning and scheduling has focused on *resource-driven* domains (such as satellite observation scheduling [7]), which emphasize optimization of resource usage in satisfying a pool of tasks. In contrast, the air operations domain has a more *goal-driven* flavor: while effective resource usage is important, the key motivation is to identify and schedule actions that will ensure attainment of stated objectives.

Objectives within the air operations domain reduce to goals of neutralizing enemy capabilities (e.g., antiaircraft capability, electricity production, communications) modeled as hierarchical networks that ground out at the level of specific targets. We provide several strategies for attacking different network types that vary in their aggressiveness, and hence resource demands. These strategies range from attacking all components in a network, to attacking a coherent subset, or an isolated node [5].

Resources (i.e., aircraft, munitions) are assigned to support prosecution of individual targets. A given type of target usually has several possible aircraft/munitions configurations. However, different configurations will have different degrees of effectiveness, and hence the numbers of resources that must be allocated to achieve the desired effect can vary with each choice. Quantities (or capacities) of different types of resources are positioned at various locations nearby or within the geographic region of interest. The set of resources assigned to fly against a given target can vary in type and, depending on availability, may either originate from multiple locations (converging on the target within a particular time interval) or recycle from the same base location (making sufficient sets of consecutive strikes on the target).

The style of planning required for this domain differs markedly from standard AI approaches. Here, the search space is dense with solutions, making it easy to find a plan that satisfies stated goals. The real challenge is to find 'good' plans rather than settling for the first available solution. While most AI planning systems seek to minimize plan size, bigger plans tend to be better in this domain since the inclusion of additional activities can increase the likelihood of achieving stated objectives. For example, eliminating more of an enemy's missile sites tends to improve the quality of a plan for neutralizing enemy attack capability. Note that maximizing plan size is not equivalent to maximizing resource usage: the planner and scheduler must still decide how to allocate available resources economically to support chosen activities.

Air operations commanders generally apportion a set of resources for a given set of high-level objectives; human planners are expected to develop solutions that maximize the likelihood of objective attainment while staying within the resource allotment. Our planning model incorporates this *apportionment perspective* into its design. In particular, initial plans seek to capitalize on all available resources; as resource problems arise, strategies are adopted that decrease resource usage.

4. Technical Approach

Our integration method builds on an incremental model of planning and scheduling that assesses resource feasibility at the level of *subplans* for the overall set of objec-

tives, using a model of *intensity* to approximate resource demand, and *adaptation* in response to scheduler feedback.

Incremental Planning and Scheduling

Within our hierarchical domain model, high-level operator choices can have a significant impact on resource requirements. However, actions with specific resource requirements do not appear until the lowest levels of a deep hierarchy. For example, the high-level decision of whether to employ a passive or more proactive approach to defending assets will greatly influence resource requirements, although the actual missions that require resources are planned at much lower levels of abstraction.

Approaches in which complete layers of a hierarchical plan are forwarded to a scheduler for resource allocation (e.g., [16]) do not provide much value in this case, since most of the plan would have to be completed before any scheduler feedback could be obtained. Instead, we developed a hybrid top-down/incremental model for planning and scheduling. The approach involves planning in standard HTN fashion down to a specified level of detail (the *decomposition layer*), and then splitting into subplans that are elaborated separately. The decomposition layer, defined implicitly in terms of specific goals, separates the higher-level strategic decisions that define overall plan structure from the planning of (mostly independent) lower-level objectives.

After completion of each subplan, the scheduler incrementally allocates resources to the new actions introduced by the subplan, taking into account the resource assignments already made for previous subplans. In the event that the scheduler is unable to produce a satisfactory resource assignment, the planner will modify one or more completed subplans to reduce resource demand, and then forward the revisions to the scheduler for appropriate adjustments to the current schedule. Once all outstanding resource problems have been resolved, the planner continues with generation of remaining subplans until completion of a full plan and schedule. With this incremental approach, the integrated plan and schedule is built in piecewise, incremental fashion, with adjustments made in response to detected resource problems

This incremental approach would be ineffective for domains in which extensive strategic dependencies link objectives. However, in our models for the air operations domain, most dependencies occur at the level of resource allocation, thus enabling the separation of the planning for individual objectives.

Intensity Models of Resource Demand

To make informed decisions about its choices, a planner requires some model of the resource impact of its decisions. Previous work on incorporating resource feasibility reasoning into hierarchical planning (e.g., [2]) has assumed the ability to determine *a priori* minimum and maximum resource requirements for individual operators at all levels of abstraction, and has used this information as decision-making guidance.

Two problems arise with approaches of this type. First, computing bounds on resource usage can be prohibitively expensive in complex domains, given the need to consider all possible goal expansions and resource allocation options. Second, for the air operations domain, the bounds obtained are likely to be weak and uninformative. This latter problem stems from two factors: the heterogeneity of the resources that might be assigned to a given subplan, and the fact that resources are physically dis-

tributed and must travel variable amounts to perform different tasks. Depending on the type of resource assigned, different numbers of resources (or different amounts of resource capacity) will be required to accomplish a particular task. The location and operating characteristics of assigned resources will dictate the overall length of time that resources must be allocated. Since in both cases, the potential variance across resource types is quite high, simple minimum (or maximum) bounds will provide overly optimistic (or pessimistic) estimates of resource demand.

Given these problems, our approach to linking planning and scheduling instead builds on a heuristic characterization of expected resource usage by a planning operator, which we refer to as an operator's *intensity*. Our work to date has explored two models for intensity, which vary both the dimensionality (*single* vs. *multi*) and the precision (*qualitative* vs. *quantitative*).

Single-dimensional Qualitative Intensity Model In this model, an operator's intensity represents a qualitative assessment of the operator's expected resource usage relative to alternatives for the same task. The air operations domain, for example, contains multiple operators for neutralizing an enemy's communication capability, ranging from taking out a single site, to destroying some select subset of communication devices, to eliminating all communication nodes. For an intensity scale of [0 10], the first operator might be ranked a 2, the second a 5, and the third a 10 to reflect their relative levels of expected resource consumption.

Multidimensional Quantitative Intensity Model This model captures expected resource usage at a finer level of granularity. Resources are grouped into functional categories intended to capture similarities in resource applicability. These groupings provide an aggregation over individual resource classes, thus simplifying the resource models inherent to the scheduler; however, the aggregation has greater detail than the single-dimensional intensity model and so would be expected to provide improved predictive value for resource usage estimation.

Within our air operations domain, for example, aircraft and munitions can be grouped according to the different types of missions in which they can be used (which is a function of target type). Our multidimensional intensity model for this domain groups 5 types of aircraft and 7 types of munitions into 4 resource dimensions. Because aircraft and munitions can be used for different types of missions, these dimensions are not mutually exclusive. This connectivity introduces additional complexity into the multidimensional intensity adaptation process, since decisions related to one dimension can impact results for others.

The multidimensional quantitative model also improves on the single-dimensional qualitative approach by employing a situation-dependent characterization of operator intensity. In particular, operator intensities are defined by a heuristic function that estimates resource demand based on the number and type of targets that an operator is expected to introduce.

The single-dimensional model has the virtue of requiring little effort to define the qualitative rankings within the underlying planning models: such rankings could be readily determinable by the knowledge engineer who develops the planning operators. In contrast, the multidimensional quantitative model requires the identification and modeling of resource abstractions. Such abstractions fall out naturally in the air operations domain but may be more problematic to define in others.

The weakness of the single-dimensional approach lies in its lack of granularity. Consider a situation with relatively low overall resource demand but where the class of resources required for a key type of action has been almost exhausted. The single-dimensional approach would not adjust strategy selection to adapt to the shortage because of the overall abundance of resources. In contrast, the multidimensional model can represent a lack of capacity for specialized groups of resources, thus enabling an adjustment in strategy selection to prefer approaches that minimize demand for the oversubscribed resource.

5. Intensity-based Adaptation

The incorporation of intensity information to guide planning occurs at the level of subplans. For a given subplan, the planner calculates a *target* intensity, denoted by I^T. This value represents the expected 'ideal' level of resource usage for a particular subplan, relative to availability and expected demand for remaining subplans. When faced with a choice among multiple applicable operators O_i for a subgoal, the intensity I^{Oi} for each is computed. Each operator is assigned a rating $Rating(O_i)$ based on how closely its intensity matches the subplan's target intensity, with the planner selecting the most highly rated operator for application. The specific definitions for the target intensity, operator intensity, and operator rating used in our work are presented below.

Adjustment of the target intensity across subplans enables the planner to adapt its strategy to match changing resource availability. The planner is provided with updates on resource availability after every interaction with the scheduler. Suppose that upon successful allocation of resources to a subplan, the scheduler's assessment of remaining resource availability indicates a shortage (excess) of remaining resources relative to the subplans yet to be generated and scheduled. By reducing (increasing) the target intensity for the next subplan to reflect this shortage (excess), the planner will be biased toward selecting operators with lower (higher) intensity values that will decrease (increase) resource consumption levels. In this way, the planner dynamically adjusts its decision-making in response to scheduler feedback.

Within this adaptive framework, different control strategies can be defined for selecting the subplan to be revised in response to scheduling problems. The experiments in this paper adopt a *chronological backoff* strategy: when the scheduler encounters a problem with a subplan, the planner reduces the target intensity for that subplan in accord with a *target intensity reduction policy* and then generates an alternative plan. This process continues until either a resource feasible subplan is found, or there is no more room for intensity reduction. In the latter case, the algorithm removes the unsuccessful subplan from the plan; if the target intensity of the previous subplan can be reduced, then planning and scheduling are tried at that lower level; otherwise, the planner continues to remove subplans until it encounters a subplan that is not yet at the minimal intensity value. From that point, it tries to plan with the lower target intensity and then restarts the generation process in the forward direction.

Below, we provide the basic definitions for target intensity, operator intensity, operator rating scheme and target intensity reduction policy for the multidimensional case, followed by their definitions for the simpler single-dimensional case.

Target Intensity I^T The target intensity for a given intensity dimension is defined in terms of the ratio of the resources available per remaining subplan to the resources

allotted originally to each subplan (assuming uniform apportionment to each); this ratio is then normalized relative to the interval of intensity values in use (namely, *[0,TopIntensity]*). More formally, let *Capacity(Ij)* be the overall capacity for resources in dimension *j* and let R_j^i be the remaining capacity for dimension *j* after the first *i* of *n* subplans have been created and scheduled. The following equation defines the target intensity I^T for the $i+I^{st}$ subplan:

$$I^T = \begin{bmatrix} I_1^T \\ \vdots \\ I_m^T \end{bmatrix} \quad where \quad I_j^T = \frac{\frac{1}{n-i} \times R_j^i}{\frac{1}{n} \times Capacity\ (I_j)} \times TopIntensi\ ty$$

Provided that resource usage remains below allotment levels, the value I_j^T will exceed *TopIntensity*. Values below *TopIntensity* indicate that planning choices should decrease demand for resources within that dimension below the original allotment level.

Operator Intensity I^{Oi} The intensity I^{Oi} of a planning operator O_i is defined by the equation:

$$I^{O_i} = \begin{bmatrix} I_1^{O_i} \\ \vdots \\ I_m^{O_i} \end{bmatrix} \quad where \quad I_j^{O_i} = \frac{ExpectedDe\,mand\,(O_i, I_j)}{\frac{1}{n} \times Capacity\,(I_j)} \times TopIntensi\ ty$$

The intensity for each dimension is defined to be the ratio of the expected resource demands introduced by the operator to the original allotment of resources for that subplan and dimension (assuming uniform allotment). For the air operations domain, the resource demands of an operator are measured in terms of the expected munitions and aircraft required to prosecute the targets associated with the operator. These estimates are calculated by summing the expected number of targets of a given type multiplied by a capacity estimate for the type.

Operator Ranking Our scheme for ranking operators according to their proximity to the target intensity values is defined by the following equations. The ranking method builds on the *intensity difference vector* $D^{Oi} = I^T - I^{Oi}$, which gives the difference between the target intensity and operator intensity vectors.

$$Rating\ (O^i) = \sum_{d_j \in D^{O_i}} Penalty\ (d_j)$$

$$Penalty(d) = \begin{cases} P^+ \times d & for\ d \geq 0 \\ P^- \times ABS(d) & for\ d < 0 \end{cases}$$

The operator rating, denoted by *Rating(O^i),* is defined to be the sum of the magnitudes in the intensity difference vector, adjusted by a *penalty factor*. In cases where the difference value d_j is positive (i.e., the operator requires fewer resources than indicated by the target intensity), the penalty is defined by P^+; in cases where $d_j<0$ (i.e., the operator is expected to use more resources than indicated by the target intensity), the penalty is defined by P^-. Through appropriate settings of the ratio of these penalty factors, different strategies can be defined that penalize resource overutilization/underutilization to different degrees. With this rating scheme, the preferred operator will be that with the lowest rating.

Target Intensity Reduction Policy In situations where the scheduler is unsuccessful in an attempt to allocate resources for a given subplan, it provides feedback to the

planner in the form of the list of problematic resources whose limited capacity have contributed to the failure. The intensity reduction policy used to adjust the target intensity for that subplan incorporates this information. In particular, each intensity dimension that includes resources from the problematic set is decreased by an amount Δ. For the experiments presented in this paper, Δ= *.25 x TopIntensity*.

Single-Dimensional Case For the single-dimensional case, the target intensity I^T reduces to

$$I^T = \frac{\frac{1}{n-i} \times R^i}{\frac{1}{n} \times Capacity} \times TopIntensity$$

The operator intensity is simply the qualitative annotation defined for the operator, while the rating is the difference between the target and operator intensities. The target intensity reduction policy consists of decreasing the current target intensity by Δ.

6. Experimental Evaluation

We conducted a series of experiments to evaluate the effectiveness of our intensity adaptation methods. Our test problem yields plans with eight subplans and 50 to 724 actions, depending upon the aggressiveness of the planning strategies applied. Experiments involved running the test problem with different resource profiles, as shown in Figure 2. The 100% profile provides just sufficient resources for the maximum plan; the profiles then decay gradually until there are insufficient resources to support the minimal plan. Additionally, the experiments employ a profile labeled BIG that contains a large amount of resources relative to the maximal plan.

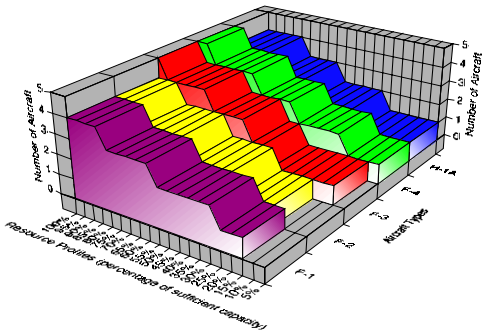


Figure 1: Experiment Resource Profiles

Generation time constitutes one important criterion for evaluating planner/scheduler behavior. Some measure of plan quality must also be considered. Otherwise, the best strategy is simply to generate the smallest plan that satisfies stated objectives: because it contains fewer activities, it will require fewer resources and so should be easier to schedule. Plan quality can be difficult to assess as it involves multiple dimensions and can be highly subjective [3]. As discussed above, air operations plans can generally be made more effective by adding more actions to them. For this reason, we use plan size as a rough indicator of plan quality.

For a baseline, we adopted a loosely coupled iterative waterfall integration of the planner and scheduler in which the planner generates complete plans and then passes

them to the scheduler for resource allocation and time-on-target assignments. If the scheduler fails to produce a feasible schedule, the process repeats with the planner performing chronological backtracking to generate alternative plans. To draw fair comparisons with the intensity-based approaches, the waterfall method considers operators in decreasing order of intensity. This strategy generally yields a plan that is close to the largest supportable for the available resources but is not necessarily optimal (i.e., chronological backtracking stops at the first solution, even though undoing an earlier operator choice might enable more aggressive subsequent choices).

Our evaluation consists of two experiments. Experiment A compares the single-dimensional and multidimensional approaches (with $P^+=P^-=1$) to the iterative waterfall. Experiment B assesses the sensitivity of the multidimensional method to the *penalty factors* P^+ and P^-. For each, we consider three performance factors: generation time, plan size, and number of planner/scheduler interactions.

Experiment A: Intensity Adaptation Evaluation

Figure 2 shows the results for Experiment A. The upper-left graph displays generation time for the three methods. As can be seen, the waterfall method requires substantially more time when resources become constrained, while the intensity-based methods perform much better. The multidimensional approach outperforms the single-dimensional approach, with the advantage increasing as resource availability drops. The upper-right graph displays the number of interactions between the planner and scheduler required to find a solution. As with generation time, these results show that the multidimensional method outperforms the single-dimensional method, and that they both are far superior to the waterfall method as resource availability decreases.

Experiment A used a scaled-down version of our air operations domain in which goals that do not involve intensity decisions are limited to a single applicable operator. This restriction was introduced to ensure that the waterfall backtracking was limited to

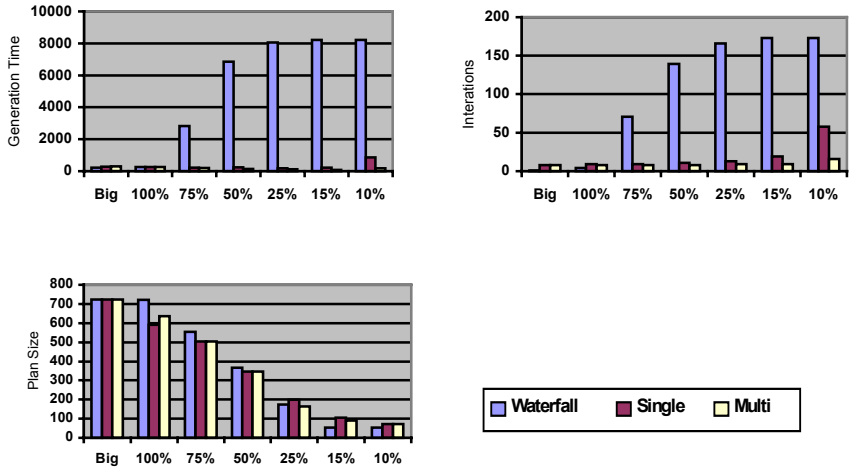


Figure 2: Comparison: Waterfall, Single-dimensional, Multidimensional Methods

precisely the same choices as the intensity adaptation methods, in essence providing the best possible comparative analysis conditions for the waterfall model. An additional experiment was run where non-intensity goals had two applicable operators. Runtimes for the intensity methods were virtually identical to those in Figure 2, since the intensity method backtracks at the level of intensity values rather than operators (hence, it is not impacted by the additional operators). In contrast, the waterfall method was unable to find a solution below the 100% resource profile after 239 trials and almost 30 hours of runtime. The waterfall method fails so badly in this larger problem because many planning decisions must be backtracked over to reach one that impacts resource usage significantly.

The waterfall approach produces slightly larger plans than the intensity-based methods for the 100% through 50% profiles; as resource availability decreases further though, it produces smaller (i.e., less aggressive) plans. In comparing runtimes, it is clear that the small increases in plan size come at the cost of an increase of several orders of magnitude in planning/scheduling time. While there is some variation between the single-dimensional and multidimensional methods, the difference is relatively small. Overall, these results show that the performance benefits realized by the multidimensional approach do not adversely impact solution quality.

Experiment B: Sensitivity to P^-/P^+

As noted above, the ratio of P^+ and P^- in the operating ranking scheme for the multidimensional approach can be adjusted to vary the penalty for overutilization/underutilization of resources relative to the established target intensity. To assess sensitivity to these values, we ran test cases with $P^+=1$ and P^- ranging from 0.5 to 4.

Figure 4 displays the results. For $P^-=4$ (and to some extent, $P^-=3$), there is a noticeable drop in plan size for the 100% through 50% profiles. For $P^-=.5$, generation times and the number of planner/scheduler interactions are appreciably higher over that same range. Such results are to be expected: when resource overutilization is

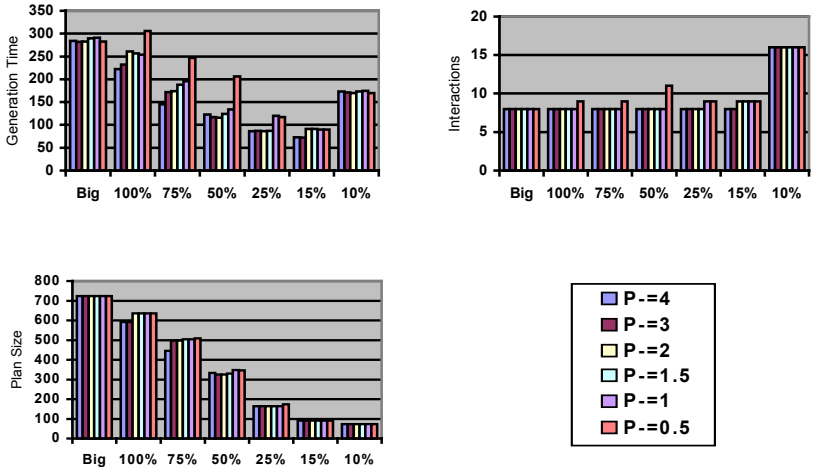


Figure 3: Sensitivity of the Multidimensional Approach to P^-/P^+

penalized relative to underutilization (i.e., $P^-/P^+ > 1$), the intensity adaptation process will be more cautious, resulting in a tendency toward smaller plans. In contrast, when resource overutilization is favored relative to underutilization (i.e., $P^-/P^+ < 1$), the intensity adaptation process will be more aggressive in its strategy selection, possibly resulting in the need for more backtracking due to overly aggressive strategy choices.

We had expected to see more dramatic variation as P^- changed but the adaptive nature of the intensity method appears to compensate for overly aggressive or weak decisions induced by large/small penalty ratios. This robustness makes the intensity adaptation approach strongly insensitive to reasonable values for parameters P^- and P^+.

7. Related Work

As mentioned earlier, much of the previous work in integrated planning/scheduling systems has been motivated by resource-driven applications. The early Hubble Space Telescope scheduling application of the HSTS system [7] provides a representative example, where a set of independent (or loosely-coupled) requests for telescope viewing time, each requiring a complex set of spacecraft actions for setup, observation, and cleanup, must be selected and sequenced for execution. Here, the overriding concern is efficient allocation of system resources, with planning decisions localized to implementation of individual tasks. The Remote Agent Planner/Scheduler [4] and the ASPEN mission planner [10] also fall into this category, as does IP3S [11], a system that integrates process planning and production scheduling in the manufacturing domain.

The REALPLAN system [14] places greater emphasis on strategic planning. Like our approach, REALPLAN partitions a problem into separate planning and scheduling components rather than solving the entire problem in a single integrated search space (see [12] for a survey of integrated search approaches). We similarly believe that such partitioning provides essential computational leverage. REALPLAN employs an iterative waterfall control model, with feedback of failure information in the most sophisticated variant. As shown in this paper, such an approach can be intractable in nontrivial domains.

The CIRCA-based planning and scheduling system described in [6] builds on an iterative waterfall model of interaction, but incorporates feedback from the scheduler to planner that is similar in spirit to our intensity adaptation approach. Based on a probabilistic state model, the planner generates *control plans* designed to prevent runtime transition to failure states. Planning relies on a specified probability threshold on states, with higher thresholds leading to consideration of fewer eventualities and simpler plans. When the scheduler is unable to meet the stated deadlines of all actions in a generated plan, it recommends a higher probability threshold to the planner for the next iteration. Similarly, when schedules underutilize resources, the scheduler suggests a lower probability threshold to enable the incorporation of additional activities.

8. Conclusions

The two intensity-based methods presented in this paper provide complementary methods for supporting effective planner/scheduler integration in domains that require

significant strategic planning. The single-dimensional qualitative approach provides a simple, easily implemented method that shows significant performance gains over waterfall-style methods. The multidimensional quantitative approach provides even better results but requires somewhat more modeling effort to operationalize.

This work represents one thrust of a larger effort to develop an integrated planning and scheduling system for management and control of large-scale enterprises [9]. Beyond the work on plan and schedule generation described here, we are developing intensity-based methods to support efficient plan and schedule repair in response to the addition or revision of objectives and changes to resource availability.

References

1. Becker, M. and S.F. Smith, "Mixed-Initiative Resource Management: The AMC Barrel Allocator", *Proc. 5th Intl. Conf. on AI Planning and Scheduling*, 2000.
2. Drabble, B. and A. Tate, "The Use of Optimistic and Pessimistic Resource Profiles to Inform Search in an Activity Based Planner", *Proc. 2rd Intl. Conf. on AI Planning and Scheduling*, 1994.
3. Gil, Y., "On Evaluating Plans", Technical Report, USC/ISI. 1998.
4. Jonsson, A., P. Morris, N. Muscettola, K. Rajan and B. Smith, "Planning in Interplanetary Space: Theory and practice", *Proc. 5th Intl. Conf. on AI Planning and Scheduling,* 2000.
5. Lee, T., "The Air Campaign Planning Knowledge Base", SRI International Technical Report, 1998.
6. McVey, C.B., E.M. Atkins, E.H. Durfee and K.G. Shin, "Development of Iterative Real-Time Scheduler to Planner Feedback", *Proc. 16th Intl. Joint Conf. on AI*, 1997.
7. Muscettola, N., S.F. Smith, A. Cesta and D. D'Aloisi, "Coordinating Space Telescope Operations in an Integrated Planning and Scheduling Framework", *IEEE Control Systems*, 12(1), 1992.
8. Myers, K.L., "CPEF: A Continuous Planning and Execution Framework", *AI Magazine, 20(4)*, 1999.
9. Myers, K.L. and S.F. Smith, "Issues in the Integration of Planning and Scheduling for Enterprise Control", *Proc. DARPA Symposium on Advances in Enterprise Control*, 1999.
10. Rabideau, G., R. Knight, S. Chien, A. Fukunaga, and A. Govindjee. "Iterative Repair Planning for Spacecraft Operations using the ASPEN System", *Proc. Intl. Symp. Of Artificial Intelligence, Robotics and Automation for Space*, 1999.
11. Sadeh, N., D.W. Hildum, T.J. LaLiberty, J. McAnulty, D. Kjenstad and A. Tseng. "A Blackboard Architecture for Integrating Process Planning and Production Scheduling, Concurrent Engineering: Research & Applications, 6(2), 1998.
12. Smith, D.E., J. Frank and A.K. Jonsson, "Bridging the Gap Between Planning and Scheduling", *Knowledge Engineering Review*, 15 (1) 2000.
13. Smith, S.F., O. Lassila and M. Becker, "Configurable Systems for Mixed-Initiative Planning and Scheduling", in *Advanced Planning Technology* (ed. A. Tate), AAAI Press, 1996.
14. Srivastava, B., S. Kambhampati and B.D. Minh, "Planning the Project Management Way: Efficient Planning by Effective Integration of Causal and Resource Reasoning in RealPlan", *Artificial Intelligence*, to appear, 2001.
15. Wilkins, D.E., *Practical Planning: Extending the Classical (AI) Planning Paradigm*, Morgan Kaufmann, 1988.
16. Wilkins, D.E. and K.L. Myers, "A Multiagent Planning Architecture", *Proc. 4th Intl. Conf. on AI Planning Systems*, 1998.

Using Abstraction in Planning and Scheduling

Bradley J. Clement[1], Anthony C. Barrett[1], Gregg R. Rabideau[1], and Edmund H. Durfee[2]

[1] Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, M/S 126-347, Pasadena, CA 91109-8099 USA
{bclement, barrett, rabideau}@aig.jpl.nasa.gov
[2] Artificial Intelligence Laboratory, University of Michigan
1101 Beal Avenue, Ann Arbor, MI 48109-2110 USA
durfee@umich.edu

Abstract. We present an algorithm for summarizing the metric resource requirements of an abstract task based on the resource usages of its potential refinements. We use this summary information within the ASPEN planner/scheduler to coordinate a team of rovers that conflict over shared resources. We find analytically and experimentally that an iterative repair planner can experience an exponential speedup when reasoning with summary information about resource usages and state constraints, but there are some cases where the extra overhead involved can degrade performance.

1 Introduction

Hierarchical Task Network (HTN) planners [4] represent abstract actions that decompose into choices of action sequences that may also be abstract, and HTN planning problems are requests to perform a set of abstract actions given an initial state. The planner subsequently refines the abstract tasks into less abstract subtasks to ultimately generate a schedule of primitive actions that is executable from the initial state. This differs from STRIPS planning where a planner can find any sequence of actions whose execution can achieve a set of goals. HTN planners only find sequences that perform abstract tasks and a domain expert can intuitively define hierarchies of abstract tasks to make the planner rapidly generate all sequences of interest.

Previous research [10, 9] has shown that, under certain restrictions, hierarchical refinement search reduces the search space by an exponential factor. Subsequent research has shown that these restrictions can be dropped by reasoning during refinement about the conditions embodied by abstract actions [3, 2]. These *summarized conditions* represent the internal and external requirements and effects of an abstract action and those of any possible primitive actions that it can decompose into. Using this information, a planner can detect and resolve conflicts between abstract actions and sometimes can find abstract solutions or determine that particular decomposition choices are inconsistent. In this paper, we apply these abstract reasoning techniques to tasks that use metric resources. We present an algorithm that processes a task hierarchy description offline to summarize abstract plan operators' metric resource requirements.

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. This work was also supported in part by DARAP(F30602-98-2-0142).

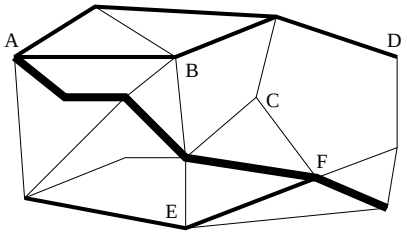


Fig. 1. Example map of established paths between points in a rover domain, where thinner edges are harder to traverse, and labeled points have associated observation goals

While planning and scheduling efficiency is a major focus of our research, another is the support of flexible plan execution systems such as PRS [6], UMPRS [11], RAPS [5], JAM [7], etc., that exploit hierarchical plan spaces while interleaving task decomposition with execution. By postponing task decomposition, such systems gain flexibility to choose decompositions that best match current circumstances. However, this means that refinement decisions are made and acted upon before all abstract actions are decomposed to the most detailed level. If such refinements at abstract levels introduce unresolvable conflicts at more detailed levels, the system will get stuck part way through executing the tasks to perform the requested abstract tasks. By using summary information, a system that interleaves planning and execution can detect and resolve conflicts at abstract levels to avoid getting stuck and to provide some ability to recover from failure.

In the next section this paper uses a traveling rover example to describe how we represent abstract actions and summary information. Given these representations, the subsequent section presents an algorithm for summarizing an abstract task's potential resource usage based on its possible refinements. Next we analytically show how summary information can accelerate an iterative repair planner/scheduler and make some empirical measurements in a multi-rover planning domain.

2 Representations

To illustrate our approach, we will focus on managing a collection of rovers as they explore the environment around a lander on Mars. This exploration takes the form of visiting different locations and making observations. Each traversal between locations follows established paths to minimize effort and risk. These paths combine to form a network like the one mapped out in Figure 1, where vertices denote distinguished locations, and edges denote allowed paths. While some paths are over hard ground, others are over loose sand where traversal is harder since a rover can slip.

2.1 Resources and Tasks

More formally, we represent each rover's status in terms of state and resource variables. The values in state variables record the status of key rover subsystems. For instance, a rover's *position* state variable can take on the label of any vertex in the location network. Given this representation of state information, tasks have preconditions/effects that we represent as equality constraints/assignments. In our rover example traveling on the arc from point A to point B is done with a *go(A,B)* task. This task has the precondition $(position{=}A)$ and the effect $(position{=}B)$.

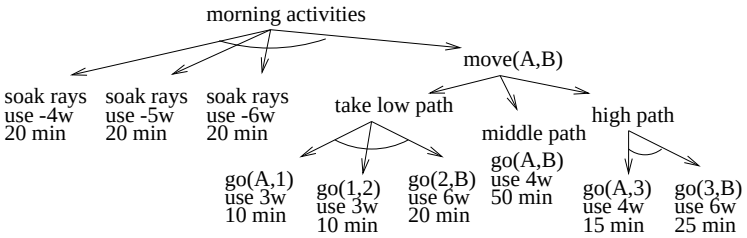


Fig. 2. AND/OR tree defining abstract tasks and how they decompose for a morning drive from point A to point B along one of the three shortest paths in our example map

In addition to interacting with state variables, tasks use resources. While some resources are only used during a task, using others persists after a task finishes. The first type of resource is *nondepletable*, with examples like solar power which immediately becomes available after some task stops using it. On the other hand, battery energy is a *depletable* resource because its consumption persists until a later task recharges the battery. We model a task's resource consumption by subtracting the usage amount from the resource variable when the task starts and for nondepletable resources adding it back upon completion. While this approach is simplistic, it can conservatively approximate any resource consumption profile by breaking a task into smaller subtasks.

Primitive tasks affect state and resource variables, and an abstract task is a non-leaf node in an AND/OR tree of tasks.[1] An AND task is executed by executing all of its subtasks according to a some set of specified temporal constraints. An OR task is executed by executing one of its subtasks. Figure 2 gives an example of such an abstract task. Imagine a rover that wants to make an early morning trip from point A to point B on our example map. During this trip the sun slowly rises above the horizon giving the rover the ability to progressively use *soak rays* tasks to provide more solar power to motors in the wheels. In addition to collecting photons, the morning traverse moves the rover, and the resultant *go* tasks require path dependent amounts of power. While a rover traveling from point A to point B can take any number of paths, the shortest three involve following one, two, or three steps.

2.2 Summary Information

An abstract task's state variable summary information includes elements for pre-, in-, and postconditions. Summary preconditions are conditions that must be met by the initial state or previous external tasks in order for a task to decompose and execute successfully, and a task's summary postconditions are the effects of its decomposition's execution that are not undone internally. We use summary inconditions for those conditions that are required or asserted in the task's decomposition during the interval of execution. All summary conditions are used to reason about how state variables are affected while performing an abstract task, and they have two orthogonal types of modalities:

- $must$ or may indicates that a condition holds in all or some decompositions of the abstract task respectively and
- $first, last, sometimes,$ or $always$ indicates when a condition holds in the task's execution interval.

[1] It is trivial to extend the algorithms in this paper to handle state and resource constraints specified for abstract tasks.

For instance, the $move(A, B)$ task in our example has a $must, first(position{=}A)$ summary precondition and a $must, last(position{=}B)$ postcondition because all decompositions move the rover from A to B. Since the $move(A, B)$ task decomposes into one of several paths, it has summary inconditions of the form $may, sometimes(position{=}i)$, where i is 1, 2 or 3. State summary conditions are formalized in [2].

Extending summary information to include metric resources involves defining a new representation and algorithm for summarization. A *summarized resource usage* consists of ranges of potential resource usage amounts during and after performing an abstract task, and we represent this summary information using the structure

$$\langle local\_min\_range, local\_max\_range, persist\_range \rangle,$$

where the resource's local usage occurs within the task's execution, and the persistent usage represents the usage that lasts after the task terminates for depletable resources.

The usage ranges capture the multiple possible usage profiles of an task with multiple decomposition choices and timing choices among loosely constrained subtasks. For example, the *high path* task has a $\langle [4, 4], [6, 6], [0, 0] \rangle$ summary power use over a 40 minute interval. In this case the ranges are single points due to no uncertainty – the task simply uses 4 watts for 15 minutes followed by 6 watts for 25 minutes. The $move(A, B)$ provides a slightly more complex example due to its decompositional uncertainty. This task has a $\langle [0, 4], [4, 6], [0, 0] \rangle$ summary power use over a 50 minute interval. In both cases the $persist\_range$ is $[0, 0]$ because power is a nondepletable resource.

While a summary resource usage structure has only one range for persistent usage of a resource, it has ranges for both the minimum and maximum local usage because resources can have minimum as well as maximum usage limits, and we want to detect whether a conflict occurs from violating either of these limits. As an example of reasoning with resource usage summaries, suppose that only 3 watts of power were available during a $move(A, B)$ task. Given the $[4, 6]$ $local\_max\_range$, we know that there is an unresolvable problem without decomposing further. Raising the available power to 4 watts makes the task executable depending on how it gets decomposed and scheduled, and raising to 6 or more watts makes the task executable for all possible decompositions.

3 Resource Summarization Algorithm

The state summarization algorithm [2] recursively propagates summary conditions upwards from an AND/OR tree's leaves, and the algorithm for resource summarization takes the same approach. Starting at the leaves, we find primitive tasks that use constant amounts of a resource. The resource summary of a task using x units of a resource is $\langle [x,x],[x,x],[0,0] \rangle$ or $\langle [x,x],[x,x],[x,x] \rangle$ over the task's duration for nondepletable or depletable resources respectively.

Moving up the AND/OR tree we either come to an AND or an OR branch. For an OR branch the combined summary usage comes from the OR computation

$$\begin{aligned}\langle [&min_{c \in children}(lb(local\_min\_range(c))),\\ &max_{c \in children}(ub(local\_min\_range(c)))],\\ [&min_{c \in children}(lb(local\_max\_range(c))),\\ &max_{c \in children}(ub(local\_max\_range(c)))],\\ [&min_{c \in children}(lb(persist\_range(c))),\\ &max_{c \in children}(ub(persist\_range(c)))]\rangle,\end{aligned}$$

where $lb()$ and $ub()$ extract the lower bound and upper bound of a range respectively. The $children$ denote the branch's children with their durations extended to the length of the

longest child. This duration extension alters a child's resource summary information because the child's usage profile has a 0 resource usage during the extension. For instance, when we determine the resource usage for $move(A, B)$ we combine two 40 minute tasks with a 50 minute task. The resulting summary information is for a 50 minute abstract task whose profile might have a zero watt power usage for 10 minutes. This extension is why $move(A, B)$ has a $[0, 4]$ for a $local\_min\_range$ instead of $[3, 4]$. Planners that reason about variable durations could use $[3, 4]$ for a duration ranging from 40 to 50.

Computing an AND branch's summary information is a bit more complicated due to timing choices among loosely constrained subtasks. Our *take x path* examples illustrate the simplest subcase, where subtasks are tightly constrained to execute serially. Here profiles are appended together, and the resulting summary usage information comes form the SERIAL-AND computation

$$\begin{array}{l}\langle[min_{c\in children}(lb(local\_min\_range(c)) + \Sigma^{pre}_{lb}(c)),\\ \ min_{c\in children}(ub(local\_min\_range(c)) + \Sigma^{pre}_{ub}(c))],\\ [max_{c\in children}(lb(local\_max\_range(c)) + \Sigma^{pre}_{lb}(c)),\\ \ max_{c\in children}(ub(local\_max\_range(c)) + \Sigma^{pre}_{ub}(c))],\\ [\Sigma_{c\in children}(lb(persist\_range(c))),\\ \ \Sigma_{c\in children}(ub(persist\_range(c)))]\rangle,\end{array}$$

where $\Sigma^{pre}_{lb}(c)$ and $\Sigma^{pre}_{ub}(c)$ are the respective lower and upper bounds on the cumulative persistent usages of children that execute before c. These computations have the same form as the Σ computations for the final $persist\_range$.

The case where all subtasks execute in parallel and have identical durations is slightly simpler. Here the usage profiles add together, and the branch's resultant summary usage comes from the PARALLEL-AND computation

$$\begin{array}{l}\langle[\Sigma_{c\in children}(lb(local\_min\_range(c))),\\ \ max_{c\in children}(ub(local\_min\_range(c)) + \Sigma^{non}_{ub}(c))],\\ [min_{c\in children}(lb(local\_max\_range(c)) + \Sigma^{non}_{lb}(c)),\\ \ \Sigma_{c\in children}(ub(local\_max\_range(c)))],\\ [\Sigma_{c\in children}(lb(persist\_range(c))),\\ \ \Sigma_{c\in children}(ub(persist\_range(c)))]\rangle,\end{array}$$

where $\Sigma^{non}_{ub}(c)$ and $\Sigma^{non}_{lb}(c)$ are the respective sums of $local\_max\_range$ upper bounds and $local\_min\_range$ lower bounds for all children except c.

To handle AND tasks with loose temporal constraints, we consider all legal orderings of child task endpoints. For example, in our rover's early morning tasks, there are three serial solar energy collection subtasks running in parallel with a subtask to drive to location B. Figure 3 shows one possible ordering of the subtask endpoints, which breaks the $move(A, B)$ into three pieces, and two of the *soak rays* children in half. Given an ordering, we can (1) use the endpoints of the children to determine subintervals, (2) compute summary information for each child task/subinterval combination, (3) combine the parallel subinterval summaries using the PARALLEL-AND computation, and then (4) chain the subintervals together using the SERIAL-AND computation. Finally, the AND task's summary is computed by combining the summaries for all possible orderings using an OR computation.

Here we describe how step (2) generates different summary resource usages for the subintervals of a child task. A child task with summary resource usage $\langle[a,b],[c,d],[e,f]\rangle$ contributes one of two summary resource usages to each intersecting subinterval[2]:

$$\langle[a, b], [c, d], [0, 0]\rangle, \langle[a, d], [a, d], [0, 0]\rangle.$$

[2] For summary resource usages of the last interval intersecting the child task, we replace $[0, 0]$ with $[e, f]$ in the $persist\_range$.

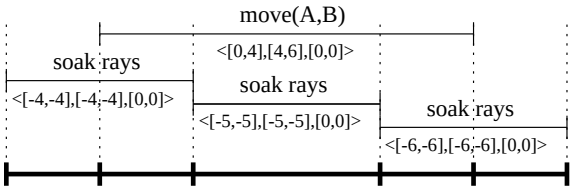


Fig. 3. Possible task ordering for a rover's morning activities, with resulting subintervals.

While the first usage has the tighter $[a, b], [c, d]$ local ranges, the second has looser $[a, d], [a, d]$ local ranges. Since the b and c bounds only apply to the subintervals containing the subtask's minimum and maximum usages, the tighter ranges apply to one of a subtask's intersecting subintervals. While the minimum and maximum usages may not occur in the same subinterval, symmetry arguments let us connect them in our computation. Thus one subinterval has tighter local ranges and all other intersecting subintervals get the looser local ranges, and the extra complexity comes from having to investigate all subtask/subinterval assignment options. For instance, there are three subintervals intersecting $move(A, B)$ in Figure 3, and three different assignments of summary resource usages to the subintervals: placing $[0, 4], [4, 6]$ in one subinterval with $[0, 6], [0, 6]$ in the other two. These placement options result in a subtask with n subintervals having n possible subinterval assignments. So if there are m child tasks each with n alternate assignments, then there are n^m combinations of potential subtask/subinterval summary resource usage assignments. Thus propagating summary information through an AND branch is exponential in the number of subtasks with multiple internal subintervals. However since the number of subtasks is controlled by the domain modeler and is usually bounded by a constant, this computation is tractable. In addition, summary information can often be derived offline for a domain. The propagation algorithm takes on the form:

- For each consistent ordering of endpoints:
  - For each consistent subtask/subinterval summary usage assignment:
    - Use PARALLEL-AND computations to combine subtask/subinterval summary usages by subinterval.
    - Use a SERIAL-AND computation on the subintervals' combined summary usages to get a consistent summary usage.
- Use OR computation to combine all consistent summary usages to get AND task's summary usage.

4 Using Summary Information

In this section, we describe techniques for using summary information in local search planners to reason at abstract levels effectively and discuss the complexity advantages. Reasoning about abstract plan operators using summary information can result in exponential planning performance gains for backtracking hierarchical planners [3]. In iterative repair planning, a technique called *aggregation* that involves scheduling hierarchies of tasks similarly outperforms the movement of tasks individually [8]. But, can summary information be used in an iterative repair planner to improve performance when aggregation is already used? We demonstrate that summarized state and resource constraints makes exponential improvements by collapsing constraints at abstract levels. First, we describe how we use aggregation and summary information to schedule tasks within an

iterative repair planner. Next, we analyze the complexity of moving abstract and detailed tasks using aggregation and summary information. Then we describe how a heuristic iterative repair planner can exploit summary information.

4.1 Aggregation and Summary Information

While HTN planners commonly take a generative least commitment approach to problem solving, research in the OR community illustrates that a simple local search is surprisingly effective [12]. Heuristic iterative repair planning uses a local search to generate a plan. It starts with an initial flawed plan and iteratively chooses a flaw, chooses a repair method, and changes the plan by applying the method. Unlike generative planning, the local search never backtracks. Since taking a random walk through a large space of plans is inefficient, heuristics guide the choices by determining the probability distributions for each choice. We build on this approach to planning by using the ASPEN planner [1].

Moving tasks is a central scheduling operation in iterative repair planners. A planner can more effectively schedule tasks by moving related groups of tasks to preserve constraints among them. Hierarchical task representations are a common way of representing these groups and their constraints. Aggregation involves moving a fully detailed abstract task hierarchy while preserving the temporal ordering constraints among the subtasks. Moving individual tasks independent of their siblings and subtasks is shown to be much less efficient [8]. Valid placements of the task hierarchy in the schedule are computed from the state and resource usage profile for the hierarchy. This profile represents one instantiation of the decomposition and temporal ordering of the abstract task's hierarchy.

A summarized state or resource usage represents all potential profiles of an abstract task before it is decomposed. Our approach involves reasoning about summarized constraints in order to schedule abstract tasks before they are decomposed. Scheduling an abstract task is computationally cheaper than scheduling the task's hierarchy using aggregation when the summarized constraints more compactly represent the constraint profiles of the hierarchy. This improves the overall performance when the planner/scheduler resolves conflicts and finds solutions at abstract levels before fully decomposing tasks.

4.2 Complexity Analysis

To move a hierarchy of tasks using aggregation, valid intervals must be computed for each resource variable affected by the hierarchy.[3] These valid intervals are intersected for the valid placements for the abstract tasks and their children. The complexity of computing the set of valid intervals for a resource is $O(cC)$ where c is the number of constraints (usages) an abstract task has with its children for the variable, and C is the number of constraints of other tasks in the schedule on the variable [8]. If there are n similar task hierarchies in the entire schedule, then $C = (n-1)c$, and the complexity of computing valid intervals is $O(nc^2)$. But this computation is done for each of v resource variables (often constant for a domain), so moving a task will have a complexity of $O(vnc^2)$.

The summary information of an abstract task represents all of the constraints of its children, but if the children share constraints over the same resource, this information is collapsed into a single *summary* resource usage in the abstract task. Therefore, when moving an abstract task, the number of different constraints involved may be far fewer depending on the domain. If the scheduler is trying to place a summarized abstract

[3] The analysis also applies to state constraints, but we restrict our discussion to resource usage constraints for simplicity.

task among other summarized tasks, the computation of valid placement intervals can be greatly reduced because the c in $O(vnc^2)$ is smaller. We now consider two extreme cases where constraints can be fully collapsed and where they cannot be collapsed at all.

In the case that all tasks in a hierarchy have constraints on the same resource, the number of constraints in a hierarchy is $O(b^d)$ for a hierarchy of depth d and branching factor (number of child tasks per parent) b. In aggregation, where hierarchies are fully detailed first, this means that the complexity of moving an task is $O(vnb^{2d})$ because $c = O(b^d)$. Now consider using aggregation for moving a partially expanded hierarchy where the leaves are summarized abstract tasks. If all hierarchies in the schedule are decomposed to level i, there are $O(b^i)$ tasks in a hierarchy, each with one summarized constraint representing those of all of the yet undetailed subtasks beneath it for each constraint variable. So $c = O(b^i)$, and the complexity of moving the task is $O(vnb^{2i})$. Thus, moving an abstract task using summary information can be a factor of $O(b^{2(d-i)})$ times faster than for aggregation.

The other extreme is when all of the tasks place constraints on different variables. In this case, $c = 1$ because any hierarchy can only have one constraint per variable. Fully detailed hierarchies contain $v = O(b^d)$ different variables, so the complexity of moving a task in this case is $O(nb^d)$. If moving a summarized abstract task where all tasks in the schedule are decomposed to level i, v is the same because the abstract task summarizes all constraints for each subtask in the hierarchy beneath it, and each of those constraints are on different variables such that no constraints combine when summarized. Thus, the complexity for moving a partially expanded hierarchy is the same as for a fully expanded one. Experiments in Section 5 exhibit great improvement for cases when tasks have constraints over common resource variables.

Along another dimension, scheduling summarized tasks is exponentially faster because there are fewer *temporal* constraints among higher level tasks. When task hierarchies are moved using aggregation, all of the local temporal constraints are preserved. However, there are not always valid intervals to move the entire hierarchy. Even so, the scheduler may be able to move less constraining lower level tasks to resolve the conflict. In this case, temporal constraints may be violated among the moved task's parent and siblings. The scheduler can then move and/or adjust the durations of the parent and siblings to resolve the conflicts, but these movements can affect higher level temporal constraints or even produce other conflicts. At a depth level i in a hierarchy with decompositions branching with a factor b, the task movement can affect b^i siblings in the worst case and produce a number of conflicts exponential to the depth of the task. Thus, if all conflicts can be resolved at an abstract level i, $O(b^{d-i})$ scheduling operations may be avoided. In Section 5, empirical data shows the exponential growth of computation with respect to the depth at which ASPEN finds solutions.

Other complexity analyses have shown that under certain restrictions different forms of hierarchical problem solving can reduce the size of the search space by an exponential factor [10, 9]. Basically, these restrictions are that an algorithm never needs to backtrack from lower levels to higher levels in the problem. In other words, subproblems introduced in different branches of the hierarchy do not interact. We do not make this assumption for our problems. However, the speedup described above does assume that the hierarchies need not be fully expanded to find solutions.

4.3 Decomposition Heuristics for Iterative Repair

Despite this optimistic complexity, reasoning about summarized constraints only translates to better performance if the movement of summarized tasks resolves conflicts and

advances the search toward a solution. There may be no way to resolve conflicts among abstract tasks without decomposing them into more detailed ones. So when should summary information be used to reason about abstract tasks, and when and how should they be decomposed? Here, we describe techniques for reasoning about summary information as abstract tasks are detailed.

We explored two approaches that reason about tasks from the top-level of abstraction down in the manner described in [3]. Initially, the planner only reasons about the summary information of fully abstracted tasks. As the planner manipulates the schedule, tasks are gradually decomposed to open up new opportunities for resolving conflicts using the more detailed child tasks. One strategy (that we will refer to as *level-decomposition*) is to interleave repair with decomposition as separate steps. Step 1) The planner repairs the current schedule until the number of conflicts cannot be reduced. Step 2) It decomposes all abstract tasks one level down and returns to Step 1. By only spending enough time at a particular level of expansion that appears effective, the planner attempts to find the highest decomposition level where solutions exist without wasting time at any level.

Another approach is to use decomposition as one of the repair methods that can be applied to a conflict so that the planner gradually decomposes conflicting tasks. This strategy tends to decompose the tasks involved in more conflicts since any task involved in a conflict is potentially expanded when the conflict is repaired. The idea is that the scheduler can break overconstrained tasks into smaller pieces to offer more flexibility in rooting out the conflicts. This resembles the EMTF (expand-most-threats-first) [3] heuristic that expands (decomposes) tasks involved in more conflicts before others. (Thus, we later will refer to this heuristic as EMTF.) This heuristic avoids unnecessary reasoning about the details of non-conflicting tasks. This is similar to a most-constrained variable heuristic often employed in constraint satisfaction problems.

Another heuristic for improving planning performance prefers decomposition choices that lead to fewer conflicts. In effect, this is a least-constraining value heuristic used in constraint satisfaction approaches. Using summary information, the planner can test each child task by decomposing to the child and replacing the parent's summarized constraints that summarize the children with the particular child's summarized constraints. For each child, the number of conflicts in the schedule are counted, and the child creating the fewest conflicts is chosen.[4] This is the *fewest-threats-first* (FTF) heuristic that is shown to be effective in pruning the search space in a backtracking planner [3]. Likewise, the experiments in Section 5 show similar performance improvements.

5 Empirical Comparisons

The experiments we describe here show that summary information improves performance significantly when tasks within the same hierarchy have constraints over the same resource, and solutions are found at some level of abstraction. At the same time, we find cases where abstract reasoning incurs significant overhead when solutions are only found at deeper levels. However, in domains where decomposition choices are critical, we show that this overhead is insignificant because the FTF heuristic finds solutions at deeper levels with better performance. These experiments also show that the EMTF heuristic outperforms level-decomposition for certain decomposition rates. In addition, we show that the time to find a solution increases dramatically with the depth where solutions are found, supporting the analysis at the end of Section 4.2.

[4] Or, in stochastic planners like ASPEN, the children are chosen with probability decreasing with their respective number of conflicts.

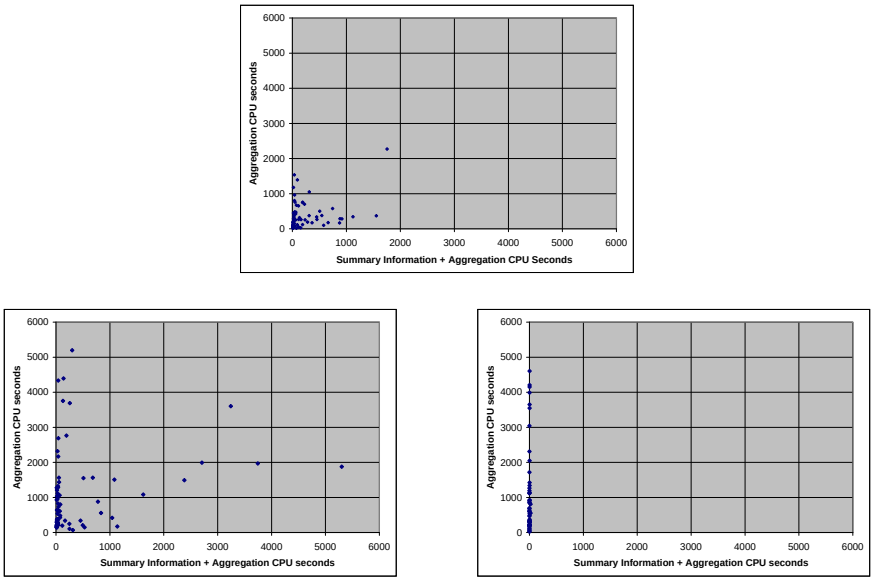


Fig. 4. Plots for the *no channel*, *mixed*, and *channel only* domains

The domain for our problems expands the single rover problem described in earlier sections to a team of rovers that must resolve conflicts over shared resources. Paths between waypoints are assigned random capacities such that either one, two, or three rovers can traverse a path simultaneously; only one rover can be at any waypoint; and rovers may not traverse paths in opposite directions. In addition, rovers must communicate with the lander for telemetry using a shared channel of fixed bandwidth. Depending on the terrain, the required bandwidth varies. 80 problems were generated for two to five rovers, three to six observation locations per rover, and 9 to 105 waypoints. Schedules ranged from 180 to 1300 tasks. Note that we use a prototype interface for summary information, and some of ASPEN's optimized scheduling techniques could not be used.

We compare ASPEN using aggregation with and without summarization for three variations of the domain. The use of summary information includes the EMTF and FTF decomposition heuristics. One domain excludes the communications channel resource (*no channel*); one excludes the path capacity restrictions (*channel only*); and the other includes all mentioned resources *mixed*). Since all of the movement tasks reserve the channel resource, we expect greater improvement in performance when using summary information according to the complexity analyses in the previous section. Tasks within a rover's hierarchy rarely place constraints on other variables more than once, so the *no channel* domain corresponds to the case where summarization collapses no constraints.

Figure 4 (top) exhibits two distributions of problems for the *no channel* domain. In most of the cases (points along the y-axis), ASPEN with summary information finds a solution quickly at some level of abstraction. However, in many cases, summary information performs notably worse (points along the x-axis). We find that for these problems finding a solution requires the planner to dig deep into the rovers' hierarchies, and once it decomposes the hierarchies to these levels, the difference in the additional time to find a solution between the two approaches is negligible Thus, the time spent reasoning about summary information at higher levels incurred unnecessary overhead. Previous work shows that this overhead is rarely significant in backtracking planners because sum-

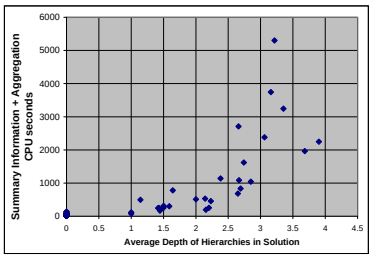


Fig. 5. CPU time for solutions found at varying depths.

mary information can prune inconsistent search spaces at abstract levels [3]. However, in non-backtracking planners like ASPEN, the only opportunity we found to prune the search space at abstract levels was using the FTF heuristic to avoid greater numbers of conflicts in particular branches. Later, we will explain why FTF is not helpful for this domain but very effective in a modified domain.

Figure 4 (left) shows significant improvement for summary information in the *mixed* domain compared to the *no channel* domain. Adding the channel resource rarely affected the use of summary information because the collapse in summary constraints incurred insignificant additional complexity. However, the channel resource made the scheduling task noticeably more difficult for ASPEN when not using summary information. In the *channel only* domain (Figure 4 right), summary information finds solutions at the abstract level almost immediately, but the problems are still complicated when ASPEN does not use summary information. These results support the complexity analysis in the previous section that argues that summary information exponentially improves performance when tasks within the same hierarchy make constraints over the same resource and solutions are found at some level of abstraction.

Figure 5 shows the CPU time required for ASPEN using summary information for the *mixed* domain for the depths at which the solutions are found. The depths are average depths of leaf tasks in partially expanded hierarchies. The CPU time increases dramatically for solutions found at greater depths, supporting our claim that finding a solution at more abstract levels is exponentially easier.

For the described domain, choosing different paths to an observation location usually does not make a significant difference in the number of conflicts encountered because if the rovers cross paths, all path choices will still lead to conflict. We created a new set of problems where obstacles force the rovers to take paths through corridors that have no connection to others paths. For these problems, path choices always lead down a different corridor to get to the target location, so there is usually a path that avoids a conflict and a path that causes one. The planner using the FTF heuristic dominates the planner choosing decompositions randomly for all but two problems (Figure 6 left).

Figure 6 (right) shows the performance of EMTF vs. level decomposition for different rates of decomposition for three problems selected from the set. The plotted points are averages over ten runs for each problem. Depending on the choice of rate of decomposition (the probability that a task will decompose when a conflict is encountered), performance varies significantly. However, the best decomposition rate can vary from problem to problem making it potentially difficult for the domain expert to choose. Our future work will include investigating the relation of decomposition rates to performance based on problem structure.[5]

[5] For other experiments, we used a decomposition rate of 20%.

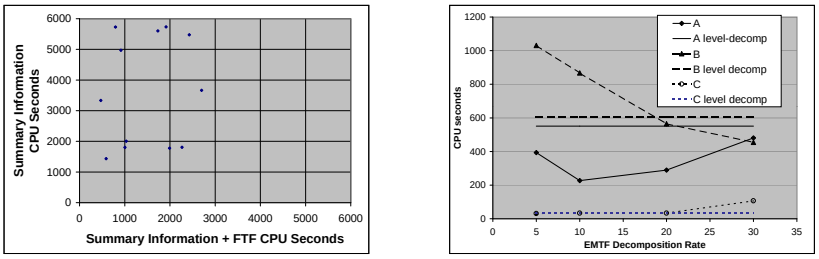


Fig. 6. Performance using FTF and EMTF vs. level-decomposition heuristics.

6 Conclusions

Reasoning about abstract constraints exponentially accelerates finding schedules when constraints collapse during summarization, and solutions at some level of abstraction can be found. Similar speedups occur when decomposition branches result in varied numbers of conflicts. The offline algorithm for summarizing metric resource usage makes these performance gains available for a larger set of expressive planners and schedulers. We have shown how these performance advantages can improve ASPEN's effectiveness when scheduling the tasks of multiple spacecraft. The use of summary information also enables a planner to preserve decomposition choices that robust execution systems can use to handle some degree of uncertainty and failure. Our future work includes evaluating the tradeoffs of optimizing plan quality using this approach as well as developing protocols to allow multiple spacecraft planners to coordinate their tasks asynchronously during execution.

References

1. S. Chien, G. Rabideu, R. Knight, R. Sherwood, B Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. Automating space mission operations using automated planning and scheduling. In *Proc. SpaceOps*, 2000.
2. B. Clement and E. Durfee. Theory for coordinating concurrent hierarchical planning agents. In *Proc. AAAI*, pages 495–502, 1999.
3. B. Clement and E. Durfee. Performance of coordinating concurrent hierarchical planning agents using summary information. In *Proc. ATAL*, pages 202–216, 2000.
4. K. Erol, J. Hendler, and D. Nau. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, University of Maryland, 1994.
5. J. Firby. *Adaptive Execution in Complex Dynamic Domains*. PhD thesis, Yale Univ., 1989.
6. M. Georgeff and A. Lansky. Procedural knowledge. *Proc. IEEE*, 74(10):1383–1398, Oct. 1986.
7. M. Huber. Jam: a bdi-theoretic mobile agent architecture. In *Proc. Intl. Conf. Autonomous Agents*, pages 236–243, 1999.
8. R. Knight, G. Rabideau, and S. Chien. Computing valid intervals for collections of activities with shared states and resources. In *Proc. AIPS*, pages 600–610, 2000.
9. C. Knoblock. Search reduction in hierarchical problem solving. In *Proc. AAAI*, pages 686–691, 1991.
10. R. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.
11. J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. Umprs: An implementation of the procedural reasoning system for multirobot applications. In *Proc. AIAA/NASA Conf. on Intelligent Robotics in Field, Factory, Service, and Space*, pages 842–849, March 1994.
12. Papadimitriou and Steiglitz. *Combinatorial Optimization - Algorithms and Complexity*. Dover Publications New York, 1998.

From Abstract Crisis to Concrete Relief

A Preliminary Report on Combining State Abstraction and HTN Planning

Susanne Biundo and Bernd Schattenberg

Department of Artificial Intelligence
University of Ulm, Germany

Abstract Flexible support for crisis management can definitely be improved by making use of advanced planning capabilities. However, the complexity of the underlying domain often causes intractable efforts in modeling the domain as well as a huge search space to be explored by the system. A way to overcome these problems is to impose a sophisticated structure not only according to tasks but also according to relationships between and properties of the objects involved. We outline the prototype of a system that is capable of tackling planning for complex application domains. It is based on a well-founded combination of action and state abstractions. The paper presents the basic techniques and provides a formal semantic foundation of the approach. It introduces the planning system and illustrate its underlying principles by examples taken from the crisis management domain used in our ongoing project.

1 Introduction

When trying to exploit planning technology for realistic applications like system support for crisis management, one of the main problems to be tackled is the complexity of the underlying domain. Not only does it cause intractable modeling efforts, a huge search space has to be explored by the system as well. Furthermore, such a system has to be flexible in the sense that mixed initiative planning has to be supported and incoming information as well as most recently arising tasks should be considered and integrated during runtime. In order to meet multiple requirements like in this case, hybrid planning approaches have to be developed to provide enough flexibility and lucidity as has repeatedly been argued by other authors as well (cf. [5] and [8]).
We introduce a planning approach for system support in the realistic and complex application domain of crisis management. It integrates hierarchical action- and state-based techniques in a consequent way by imposing hierarchical structures on both operators and states. The hierarchical concept is partly adopted from traditional hierarchical task network (HTN) planning (cf. [4]). Therefore, basic notions like primitive and abstract tasks as well as methods for decomposing the latter stepwise into primitive ones are among the core concepts. However, tasks do show pre- and postconditions –like operators do in classical state-based planning– on every level of abstraction. This provides the flexibility to make use of state-based planning techniques by introducing additional tasks when trying to establish missing preconditions, and enables the system to integrate incoming tasks on any level of abstraction at any time. This is done by allowing for so-called *decomposition axioms*, which are defined as part of the domain model. The planning approach is based on a formal semantics, which relies on previous work on logic-based planning [20]. Our formalism builds upon the semantics for basic

STRIPS-like operators and extends the formalism to abstract tasks and decomposition methods.
A first prototype implements our integrative approach. We use an object-oriented programming paradigm, thereby exploiting object-oriented structures and mechanisms to efficiently deal with the hierarchy of planning objects and their properties. According to our experience with this first implementation, the system is currently being completed and extended towards several directions.
The paper is organized as follows. In Section 2 we introduce the formal semantics. The application domain –a mission of the German Federal Agency for Technical Relief at a flood– is briefly introduced in Section 3. Section 4 describes our planning method and illustrates the techniques by means of examples taken from the crisis management domain of Section 3. In Section 5, we shortly report on the implementation and the lessons learned from this experiment. Section 6 is devoted to related work and finally we conclude with some remarks in Section 7.

2 Formal Framework

The Logical Language: The semantics of our planning approach is based on a many-sorted first-order logic. The logical language $L = (Z,R_r,R_f,C,V)$ consists of a finite set of *sort symbols* Z, Z^* indexed families of finite disjoint sets of *rigid* and *flexible relation symbols* (R_r and R_f, resp.), a Z indexed family of disjoint sets of *constants*, and a Z indexed family of disjoint sets of *variables*. Formulae over L are built as usual. The formal planning language is obtained by extending L by O, T, and E. O and T are Z^* indexed families of finite disjoint sets of *operator* and *task symbols*, respectively. For all $\bar{z} \in Z^*$ the sets $R_{r,\bar{z}}$, $R_{f,\bar{z}}$, $O_{\bar{z}}$, and $T_{\bar{z}}$ are supposed to be mutually disjoint. E denotes a Z^* indexed family of so-called *elementary operation symbols*. It provides for each flexible relation symbol R a so-called *add-operation* +R as well as a *delete-operation* –R.
As for the semantics, we adopt some essential features of the planning formalisms introduced in [19] and [20], which are based on programming and temporal logics, respectively.
Following a *state-based* planning approach, we use operators and tasks to take us from one state to another. The flexible symbols provided by our planning language are used to express the changes caused by these state transitions. Consequently, we introduce states as interpretations of the flexible symbols.

States and State Transitions: For a logical language $L = (Z,R_r,R_f,C,V)$ a *model* denotes a structure $M = (D,S,I)$, where D is a Z indexed family of *carrier sets*, S is a set of *states*, and I is a (state-independent) *interpretation* that assigns elements of the respective carrier sets to constants and a relation of appropriate type to each rigid symbol. As usual, sort preserving *valuations* $\beta : V_z \rightarrow D_z$ are used for variables. Given a model $M= (D,S,I)$, an atomic formula $\mathrm{R}(\tau_1, ..., \tau_n)$ is *valid* in a state $s \in S$ under a valuation β denoted by $s \models_{M,\beta} \mathrm{R}(\tau_1, ..., \tau_n)$ according to the following definition.

For $\mathrm{R} \in R_r$: $s \models_{M,\beta} \mathrm{R}(\tau_1, ..., \tau_n)$ iff $(I_\beta(\tau_1), ..., I_\beta(\tau_n)) \in I(\mathrm{R})$
For $\mathrm{R} \in R_f$: $s \models_{M,\beta} \mathrm{R}(\tau_1, ..., \tau_n)$ iff $(I_\beta(\tau_1), ..., I_\beta(\tau_n)) \in s(\mathrm{R})$

Based on these definitions, validity of complex formulae is defined as usual.

Now we are ready to turn from *states* to *state transitions*. To this end, we first assume that our models are *natural* ones [19]. This means, the carrier sets are supposed to be finite and we restrict the set of states to those, which assign finite relations to the symbols in R_f. Furthermore, for each flexible symbol R $\in R_{f,\bar{z}}$, $\bar{z} = z_1, ..., z_n$, two functions d-R : $D_{z_1} \times ... \times D_{z_n} \rightarrow S \times S$ and a-R : $D_{z_1} \times ... \times D_{z_n} \rightarrow S \times S$ are defined as follows.

s d-R$(d_1, ..., d_n)$ s' iff s'(R) = s(R) $- \{(d_1, ..., d_n)\}$ and s'(R') = s(R') for R' $\neq$ R
s a-R$(d_1, ..., d_n)$ s' iff s'(R) = s(R) $\cup \{(d_1, ..., d_n)\}$ and s'(R') = s(R') for R' $\neq$ R.

Given a natural model, for any two states s and s' there exists a finite sequence of a-... and d-... function operations $op_1 ... op_n$ such that s $op_1 \circ ... \circ op_n$ s', where $\circ$ denotes functional composition [19].

Elementary Operations: Based on the definitions of a-... and d-... functions on states, we can now define the semantics of the elementary operations E of our planning language as follows. Given a model $M = (D,S,I)$ and a valuation β, a pair of states (s, s') satisfies an elementary operation +R(τ_1, ..., τ_n) according to

$$(s, s') \models_{M,\beta} +\mathrm{R}(\tau_1, ..., \tau_n) \text{ iff } s \ a\text{-R}(I_\beta(\tau_1), ..., I_\beta(\tau_n)) \ s'$$
$$(s, s') \models_{M,\beta} -\mathrm{R}(\tau_1, ..., \tau_n) \text{ iff } s \ d\text{-R}(I_\beta(\tau_1), ..., I_\beta(\tau_n)) \ s'$$

This means, elementary operations represent single state transitions.
We finally adopt from [19] the concept of *weakest preconditions* (wp) w.r.t. elementary operations. Let φ be a formula which contains only variables that are distinct from those occurring in τ_1,...,τ_n. The *weakest precondition* of φ w.r.t. +R($\tau_1, ..., \tau_n$) is the formula resulting from φ when replacing all atomic sub-formulae R(σ_1,...,σ_n) by [($\tau_1 \neq \sigma_1 \vee ... \vee \tau_n \neq \sigma_n$) $\rightarrow$ R(σ_1,...,σ_n)]. wp(φ, –R($\tau_1, ..., \tau_n$)) results from φ by replacing all atomic sub-formulae R(σ_1,...,σ_n) by [R(σ_1,...,σ_n) $\wedge$ ($\tau_1 \neq \sigma_1 \vee ... \vee \tau_n \neq \sigma_n$)].

Operators and Invariants: Given a planning language $P = (Z,R_r,R_f,C,V,O,T,E)$, an *operator* (*primitive task*) is a triple (O($\bar{\mathrm{x}}$), prec, $\bar{\mathrm{e}}$), where O is an operator symbol, $\bar{\mathrm{x}} = \mathrm{x}_1...\mathrm{x}_n$ is a list of variables, prec is a formula over $L = (Z,R_r,R_f,C,V)$, and $\bar{\mathrm{e}} = \mathrm{e}_1...\mathrm{e}_m$ is a (finite) sequence of elementary operations from E. For a given model M= (D,S,I) and a valuation β, this operator transforms a state s into a state s', denoted by $(s, s') \models_{M,\beta}$ (O($\bar{\mathrm{x}}$), prec, $\bar{\mathrm{e}}$), iff $s \models_{M,\beta}$ prec (the operator is applicable in s) and s $op_1 \circ ... \circ op_m$ s' where op_i is the a-... resp. d-... function corresponding to e_i for $1 \leq \mathrm{i} \leq \mathrm{m}$. The operator *generates* a formula post over L if in addition $s \models_{M,\beta}$ wp(post,$\bar{\mathrm{e}}$). The weakest precondition of a formula φ w.r.t a *sequence* of elementary operations is generated according to a straightforward extension of the above definition.
Before finally defining *tasks* and *methods*, we introduce the notion of *invariant* in order to extend the *generation* of formulae from single operators to operator *sequences*.
For a given model M= (D,S,I) and a valuation β, a formula φ is *invariant* against an operator (O($\bar{\mathrm{x}}$), prec, $\bar{\mathrm{e}}$) iff for all states s and s' with $(s, s') \models_{M,\beta}$ (O($\bar{\mathrm{x}}$), prec, $\bar{\mathrm{e}}$) : if $s \models_{M,\beta} \varphi$, then $s' \models_{M,\beta} \varphi$.
A formula post is *generated* by a sequence $\mathrm{O}_1...\mathrm{O}_n$ of operators iff it is generated by some O_i ($1 \leq \mathrm{i} \leq \mathrm{n}$) and is invariant against each O_j ($\mathrm{i} < \mathrm{j} \leq \mathrm{n}$).

Tasks and Methods: Given a planning language $P = (Z,R_r,R_f,C,V,O,T,E)$, a *task* is a triple (T($\bar{x}$), prec, post), where T is a task symbol, $\bar{x} = x_1...x_n$ is a list of variables, and prec and post are formulae over $L = (Z,R_r,R_f,C,V)$. For a given model $M = (D,S,I)$ and a valuation β, the task transforms a state s into a state s', denoted by $(s, s') \models_{M,\beta}$ (T($\bar{x}$), prec, post) iff $s \models_{M,\beta}$ prec and $s' \models_{M,\beta}$ post and there exist a finite sequence $s_1...s_n$ of states and a finite sequence $O_1...O_{n-1}$ of operators, where $s = s_1$, $s' = s_n$, $(s_i, s_{i+1}) \models_{M,\beta} O_i$ for all $1 \leq i < n$, and $O_1...O_{n-1}$ generates a formula post$'$ such that $s_n \models_{M,\beta}$ post$' \rightarrow$ post. The task *generates* a formula post$''$ iff in addition $s_n \models_{M,\beta}$ post $\rightarrow$ post$''$.
The hierarchical structure of planning domains is reflected in two ways. First of all so-called *methods* are used to specify how an abstract task can be subdivided into a set of (primitive) subtasks, like it is usually done in HTN planning. Secondly, a hierarchy is imposed on the formulae used to express the pre- and postconditions of primitive and non-primitive tasks. To this end, user-defined *decomposition axioms* of the form $\varphi \leftrightarrow [\psi_1 \vee ... \vee \psi_n]$ specify how an abstract condition φ can be refined into a more concrete one, each ψ_i being a possibility to do so.
A method $\{$(T($\bar{x}$), prec, post), $\mathcal{T}\}$ is given by a task and a set $\mathcal{T}$ of task sequences. For each such sequence $t_1...t_n$ the t_i may be primitive or non-primitive. A method is called *legal* iff each task sequence $t_1...t_n \in \mathcal{T}$ is a legal decomposition of the task.
For a given model $M = (D,S,I)$ and a valuation β a task sequence $t_1...t_n$ is a *legal decomposition* of a task (T($\bar{x}$), prec, post) iff the task sequence transforms a state s into s' such that for the precondition prec$'$ of task t_1 $s \models_{M,\beta}$ prec$' \rightarrow$ prec and the sequence $t_1...t_n$ generates a formula post$'$ such that $s' \models_{M,\beta}$ post$' \rightarrow$ post.
The above mentioned axioms are thereby used to justify legality of decompositions when specifying methods. Illegal decompositions can be detected during compilation of the domain.

3 The Application

Our planning domain is crisis management as being provided by organizations like THW. The German *Technisches Hilfswerk* is a governmental disaster relief organization that provides technical assistance at home as well as humanitarian aid abroad. Their mission within the flood disaster at the river "Oder" in July 1997 is used in our ongoing project to build a first realistic domain model for the planner. In the following, examples from this domain will be used to demonstrate our approach. The tasks of the THW are rich and widespread, they cover all aspects of crisis management, ranging from first measures after a hazardous event to long term supplies after clearing some disaster area. Therefore, Figure 1 only shows a relative small part of the complex task hierarchy, and most task networks are depicted as single, more "self-explanatory" actions.
The most abstract task is named `flood-disaster`. It comprises a management and communication task to determine which areas are endangered to what level. Furthermore, the logistics and supplies have to be installed, e.g. quarters for the relievers to be set up. The evacuation and the securing of the embankment are the most crucial sub-tasks in reality, and during all the activities, the relievers have to clear the area continuously, i.e. to check damaged buildings, to remove perished animals, etc.
In our examples we will focus on the evacuation task, which consists of two sub-tasks: one informs the population about the relief measures, the second brings people to safe

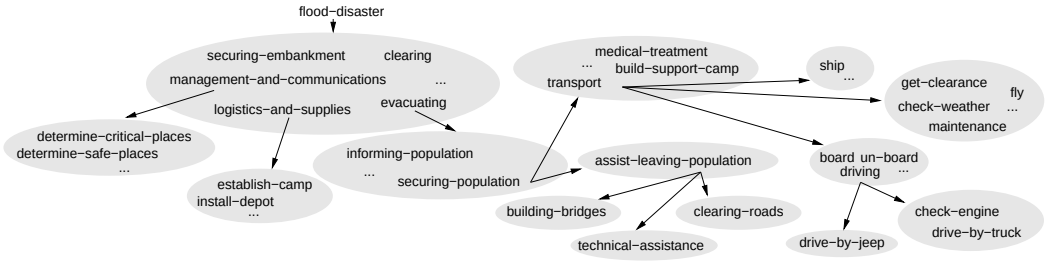


Figure 1. Task hierarchy in the *flood disaster* scenario, ellipses represent task networks

areas. The methods for latter define two expansions, depending on the initial situation. The relievers can help the people to leave the endangered area by themselves, or the circumstances might require the population to be moved by the THW.

4 Combining hierarchical action- and state-based planning

Our planning approach flexibly combines classical HTN and classical state-oriented POCL planning based on the formal framework introduced in Section 2. The prototype implements a simple top-level algorithm comparable to [16, p. 374], which is basically a classical nonlinear planning algorithm with decomposition of abstract tasks as an additional plan modification step. Although it looks very similar to those used by existing hybrid planning systems (see section 6), we will use it to outline the underlying principles in the sub-routines. First we will focus on the closing of open preconditions. In order to enable the planner to reason about the plans' causal structures and dependencies at all levels of abstraction, complex tasks do carry preconditions and effects like the operators do. For the time being they are assumed to be conjunctions of positive and negative literals.
While the relation between abstract and primitive tasks is given by a number of methods as in classical HTN planning, in our approach relations between the respective preconditions and effects are specified by the decomposition axioms. The example in Figure 2 shows two methods for the expansion of the abstract transport task in the evacuation context. The ordering constraints represent all possible sequences of subtasks, sort information for the variables is given in the task definitions.
One of the decomposition axioms that will be applied in the respective expansion steps, will e.g. look like this (assuming the intuitive subsort relationships):

$$\begin{aligned}\text{At}(\text{Unit } u, \text{Location } l) \leftrightarrow [\ &\text{Standing-at}(\text{Vehicle } u, \text{Location } l, \text{Road } r) \vee \\ &\text{Aircraft-at}(\text{Aircraft } u, \text{Location } l, \text{Height } h) \vee \\ &(\text{At}(\text{Container } c, \text{Location } l) \wedge \text{In}(\text{Container } c, \text{Unit } u)) \vee \dots\ ]\end{aligned}$$

The specified decomposition axioms together with the sort and subsort definitions, represent a hierarchy on the relations and objects in the domain. We can make use of this knowledge when closing open preconditions with tasks on different levels of abstraction. When some (possibly abstract) effect of a task is needed to establish the precondition of another, the planner can provide this by choosing some –according to

```
method m_1
  expands transport(?passengers,?from,?to,?by)
  vars    ?road Road
  nodes   (1:board(?passengers,?from,?by))
          (2:driving(?by,?from,?to,?road))
          (3:un-board(?passengers,?from,?by))
          ...
  order   1<2, 2<3
  causal  1--in(?passengers,?by)--2

  binding -
```

```
method m_2
  expands transport(?passengers,?from,?to,?by)
  vars    ?tower Tower
  nodes   (1:get-clearance(?from,?tower,?by))
          (2:check(?by))
          (3:board(?passengers,?by))
          (4:fly(?by,?from,?to,?tower))
          ...
  order   1<4, 2<4, 3<4
  causal  1--cleared(?by)--4,
          2--checked(?by)--4,
          1--in(?passengers,?by)--4
  binding -
```

Figure 2. Example for a method definition

the decomposition axioms– suitable tasks in the partial plan to close the open condition. We then add causal links like in classical non-hierarchical POCL planning to represent causality. But no establisher may be identified, even in the initial state. In this case the planner can introduce a suitable establisher for the open condition from the domain description. Figure 3 shows the planning process in such a situation.

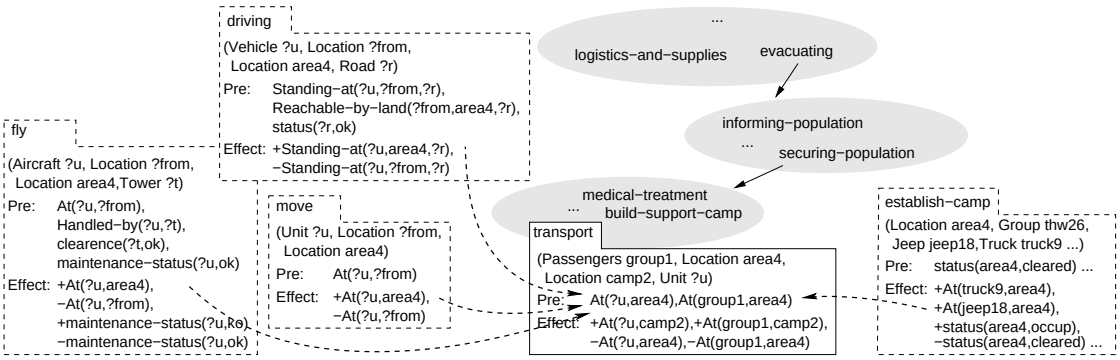


Figure 3. Closing an open precondition along the condition hierarchy.

The abstract need for an arbitrary THW unit to be present in the evacuation area can be fulfilled by any of the tasks shown. The first task `move` is a "classical" candidate. Its sub-task `fly` and the `establish-camp` action qualify in our system, because of aircraft and jeeps being sub-sorts of the abstract sort `unit`. The driving task establishes a more specialized effect than the precondition needs in two ways. Not only vehicles are more special objects, but also the relation `Standing-at` is more concrete than `At` (see decomposition axiom above).

At this point, search control has more choices to investigate, some of which might specialize the involved objects too early, an effect sometimes called *hierarchical promiscuity*. But on the other hand, the commitment to a less abstract establisher can rule out inconsistent solutions at an early stage. We may use the driving task for closing the condition at this point and add a variable assignment for the "downcast" of the unit

in the transportation task. Later in plan generation we might find out, that there is no road to the evacuation area anymore, and the planner has to backtrack and focus on solutions with aircraft.

We note, that especially at this point the search strategy plays a crucial role, i.e. when to insert new tasks. Currently, we work on an extension to the algorithm, that is looking for invariants in expansions. If an open precondition is invariant against all tasks expanded so far, it is obvious, that the planner has to insert a new task. But if ther are tasks in the current plan, against which the condition is not invariant, it is a promising strategy to enforce their expansion. The rationale behind this strategy is to check, whether the expansions in which the desired effect might manififests eventually result in consistent solutions.

Now assume, the abstract movement is chosen for establishing the condition, and a causal link with the label `At(?u,area4)` is inserted. Furthermore, let the planner decide to expand the transport task according to the above definition in method `m_1` into the task network describing a transport by land vehicles. As in classical HTN planning, the specified network substitutes the expanded task in the net, respecting existing orderings and variable bindings, but we cannot update the causal links, because the less abstract tasks show more concrete, and hence syntactically not equivalent, conditions. Our solution lies in the decomposition axioms, according to which we distribute the abstract effects and conditions under the tasks of the expanded network. This means, the decomposition axioms are used to inherit causal links from an abstract level to a more concrete one. Figure 4 shows the result: the passengers are boarded on some vehicle, driven to the camp and then un-boarded again. The more abstract link carried

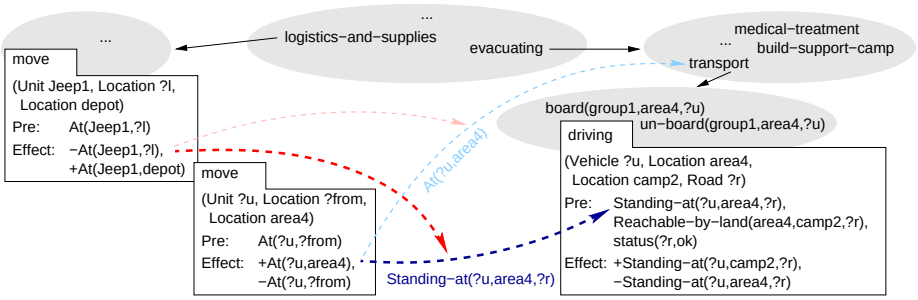


Figure 4. Handling task interactions on different abstraction levels

the `At` relation between a vehicle `?u` and the location `area4`. The decomopsition axioms justify a specialization of this link into `Standing-At` for vehicles. Please note, that the vehicle boarding task may carry the same precondition, which leads to a second inherited causal link derived from the abstract causal relation. Furthermore, when the abstract `move` task is specialized, it has to be checked against the decomposition axioms, whether the newly introduced causal links are inheritable or not. If not, the plan has to be considered inconsistent.

Now we come to threat handling, where the system can make use of the decomposition axioms like it did for condition establishment. Conflicts can be detected and resolved

between arbitrary expansion levels, as the negation of an abstract condition implies the negation of every concrete one specified in the decomposition axioms, and a negation of one of the concrete conditions threatens the abstract one. This mechanism guarantees correct solutions when using the standard POCL conflict resolution strategies at any time the algorithm chooses to check for threats. In addition, besides orderings and non-codesignation, expansion becomes a reasonable threat resolution mechanism. Due to a finer granularity, the conflicting effect might turn out to be harmless at the primitive operator level where the conflicting tasks may overlap.
The example in Figure 4 shows the expansion of a support procedure with a second movement task, and how it interferes with an established condition. The light line indicates the conflict at an abstract level, the dark one does so for the more concrete link. The negated effect can be specialized in a way that makes the plan inconsistent. A simple non-codesignation or some ordering constraint can solve this situation, for a conflict involving several layers of abstraction.

5 Implementation

We implemented a first prototype of the planning system in Java. It integrates task decomposition with state-based planning techniques for conflict handling and closing of preconditions. The main algorithm, which is briefly described in Section 4, performs non-deterministic steps according to a very simple strategy that first tries to close open preconditions, then resolves threats and binds variables, and with least priority expands complex tasks. This rather simple procedure is able to perform a systematic planning strategy like it can be found in classical nonlinear planners: As a last choice in closing preconditions, binding variables, and resolving threats, it tries to expand the respective task. In a least commitment fashion it tries to develop the plan at the most abstract level. The upcoming next version of the system will use a more flexible top level routine which determines the appropriate sequence of plan manipulating steps by analyzing the visited and expected plan space, thereby projecting causal interactions. As already suggested above, we found it e.g. very useful to concentrate expansion within the conflicting tasks to rule out inconsistent solutions at an level as early as possible. However, we note that in the first experiments the proposed modeling approach seems to lead to more “benign” domain models in terms of efficient task hierarchies, that prune large parts of the non-useful search space quite efficient.
To increase the system’s performance, we use a conservative algorithm for manipulating a global plan structure representing the expanded networks. By doing so, we have the additional advantage to automatically bookkeep the performed expansion steps as well as all other choices made by the algorithm (cf. decomposition links in [24]). When looking for an appropriate effect to close an abstract precondition, the system can easily inspect already expanded abstract tasks and follow their decomposition to less abstract levels.
The algorithm allows recursive task expansion schemata to model loops. If for every recursive task a terminating method is provided and if an appropriate search algorithm is used (here: iterated deepening) then the recursion is harmless with respect to program termination and “increasing the incompleteness” of the planning process. These loops are very useful in the presented examples, e.g. evacuation has to be performed until all persons safe, although recursion handling still requires improvement (cf. future work in section 7).

6 Discussion and Related Work

Hierarchical task network (HTN) planning as described and analyzed in [3] is the basis for systems like O-Plan, UMCP and Shop.
In contrast to our approach, which makes use of state abstractions in condition achievement, abstract tasks in O-Plan [2] do not carry preconditions and effects. Instead, the system relates conditions of primitive operators over different levels in the plan generation process by introducing *condition types* in the abstract expansion schemes [21]. These types specify how conditions of the tasks in the expansion can be achieved: by the effect of a task that is *(a)* inside or outside the current expansion and *(b)* introduced at the current plan generation level, above, or below. Please note, that this technique requires the domain encoder to structure the task hierarchy very carefully as methodologically it's pruning works rather on the system's search space structure than on that of plan space. Compared to O-Plan, UMCP [4] is a much more puristic implementation. The search space is constraint pruned, down to the most concrete operator level, where the typical conditions are introduced. Both systems merely rely on action abstraction.
A new direction in the HTN paradigm is given by the Shop system [14], that proposes ordered task decomposition, using if-then-else cascades in method selection. The main idea is to plan all tasks in the order they are later executed. This enables the system to deduce complete state descriptions, beginning with the initial state. The developers met the criticism on their linearity assumption with a modified system, called M-Shop [15] which can handle planning problems with parallel goals in the initial task. Many realistic domains may meet this partial linearity property, the crisis management domain in our work, however, does not as task execution itself is higly distributed and the execution order for most tasks is not known in advance.
Planning using state abstraction was the earliest form of hierarchical planning in linear planning systems. Nonetheless, the Abstrips system [17] is still discussed [7], and has influenced many modern planners. Classical state abstraction works by deleting certain sets of preconditions, thereby defining "criticality levels" for each of which the system plans in a classical manner. Alpine in [9] automatically generates these levels, building abstraction hierarchies with "ordered monotonicity", i.e. detailed action levels do not interfere with more abstract established conditions. Similar work in the context of nonlinear planning has been done by Yang in the Abtweak planner [23].
Exploiting object-oriented formalisms or state abstraction is a comparatively new technique in planning. Semantic foundations for "real" object oriented approaches can be found in the literature, ranging form reasoning about object database models in the style of terminological logics [1] to specification oriented work [18]. [6] uses plans as object methods for an hybrid reactive robot controller, mapping incoming percepts on partially specified object templates as plan selection criteria. More related to our view is that of object centered planning [13], where objects are organized in static and dynamic sorts. Each instance of a dynamic sort has its own local state which is defined by a set of predicates. Consequently, predicates are owned by exactly one sort, the key attribute of the predicate, and thereby becoming static or dynamic themselves. For all sorts legal local states are specified, and over their transitions again operators. OCLh [12] extends this formalism to action abstraction by introducing a sort hierarchy, in which dynamic predicates are inherited from super-sorts. So-called guards play the role of pre and postconditions of objects transition sequences that build the semantics

for abstract tasks. The planning algorithm in this framework repeats an expand then make-sound cycle: after expanding one level of task networks, the system is checking for inconsistencies and repairing them. Although our state abstraction is similar (and so far yet simple, compared to "full" object oriented systems), we can handle the refinement of objects and predicates, likewise, and are not restricted to a fixed planning strategy.

Integrating state-based nonlinear planning capabilities, i.e. reasoning about operator interactions and inserting new plan steps/tasks in the fashion of e.g. UCPop [11], into an action abstracting system promises many advantages. As Estlin, Chien and Wang point out in [5] it adds the strengths of both, at the same time softening their weak points. This is reflected in the modeling process: task networks more naturally represent hierarchy and modularity and enable the user to represent domains in an object oriented form which easier to write and reason about. Decomposition rules can refer to either low- or high-level forms of a particular object or goal, as the information pertaining to specific entities is contained in smaller, more specialized rules. The drawback of this technique is that "inter-modular constraints" [5], i.e. exceptions or special cases in action execution, cannot be represented adequately, which often leads to overly-specified reduction rules. This can be seen in the example in figure 3, where classical hierarchical planners would introduce expansion schemes for every kind of support task to be ordered before the evacuation. Operator based techniques on the other hand help encoding implicit constraints, as their kind of plan refinement is more general and provides more compact representations. In addition, it brings with it an early detection of inconsistencies at an abstract level, together with means of resolving the conflicts. But using solely operators, certain aspects are difficult to represent (for a discussion about the expressive power of HTN planning, see [3]). The advantages of a natural mixed domain knowledge representation are obvious, although difficult to evaluate quantitatively: "[it is] easier to encode the initial knowledge base, fewer encoding errors occur [...], and maintenance of the knowledge base is considerably easier."[5]

Such hybrid systems had been watched suspiciously a long time, because the planning paradigms were considered to be conflictive. New AI textbooks present this approach in the style of state abstraction planning in [23], i.e. the abstract tasks carry preconditions and effects from a subset of the less abstract tasks. Yang suggests in [22] to keep hierarchical models restricted in such a way, that in every reduction schema there is one task carrying the main effects of the network and hence those of the associated abstract task. In such domains the downward solution property holds as a basis for effective search space reduction. A similar approach is presented by Russell and Norvig [16], who allow distribution of conjuncts of conditions among the sub-tasks of the network. One of the very few existing systems is DPOCL [24], mentioned before with its introduction of decomposition links to record decisions during planning. DPOCL decomposes abstract tasks into networks with additional initial and final steps which carry the conditions of the abstract tasks. Some of the techniques used there raise the crucial question of user intent. The system prunes unused steps and takes condition establishers from every level of abstraction, even from sub-tasks of potential establishers. The problem of when to insert new tasks, and where to use decomposition rules only, is very hard to solve, as it depends in part on the modeller's intention. So far, we have provided the system a switch for explicitly not inserting new tasks in precondition achievement, as well as an output, indicating the inserted tasks Moreover, premature insertion of new tasks may lead to non-optimal short plans, but we postpone this problem for this

time as a matter of "good" search strategies, like it is solved for classical state-based nonlinear planners –but of course it will be tackled in the future.

Closely related to our approach is the work of Kambhampati [8]. He integrates HTN planning in a general framework for refinement planning, thereby making use of operator based techniques. This unified view should help making use of recent progress in planning algorithms, e.g. by giving propositional encodings for SAT based planners [10]. In his view, the algorithm uses reduction schemes where available, and primitive actions otherwise. Causal interaction is analyzed also at the abstract level, and refined by a mapping of conditions and effects of abstract tasks on conditions and effects in its sub-tasks. Abstract conditions are closed by phantom establishers that are identified at a later stage, while our algorithm just "waits" if no suitable task is less abstract enough. Conflict detection and resolution can only be done at the primitive level, as in contrast to our methodology, there is no "vertical" link between causalities in the different levels of abstraction. Kambhampati addresses user intent by defining a subset of abstract effects explicitly for condition establishment, and by explicit representing the incompleteness of scheme definitions. For the latter, a specific predicate prevents insertion of new steps.

Another aspect of hybrid planning to mention is its relevance to the relative new area of mixed initiaive planning. Only small modifications to hybrid planning algorithms allow the user to propagate decisions as commitments to the planner, including insertion of new tasks (many technical problems concerning systematicity of the system, etc. are of course beyond the scope of this paper). The resulting system benefits from our state abstraction technique, because the intermediate results, which are the basis for user interaction, become more usable in two ways: *(a)* The explicit representation of causal interactions is intuitive, even for abstract tasks, and *(b)* all modifications can be done at an arbitrary level of abstraction. An example might be an abstract plan with transport tasks, for some of which a human user can decide – on the basis of the plan developped so far – not to be performed by aircraft. He can introduce at this level constraints to choose land vehicles.

7 Conclusions and Future Work

We have introduced a planning approach that integrates hierarchical task networks and state-based POCL planning techniques by imposing hierarchical structures on both tasks and state descriptions. Tasks on all abstraction levels are extended by pre- and postconditions, which enable the flexible integration of hierarchical decomposition and nonlinear planning. A formal semantics of the approach provides the notion of legal decomposition, among others. It is an essential means to ensure that during domain modelling tasks and state abstractions are defined in a mutually consistent way. A planning system has been presented, which implements this integrative planning approach. It will be used to flexibly generate mission plans for environmental disasters.

Future work will, among others, be devoted to even further exploit the object-oriented implementation paradigm and to the implementation of a more flexible search strategy. Furthermore, the example domain strongly demands resource reasoning, especially time, and a specialized loop mechanism.

References

1. D. Calvanese, G. De Giacomo, and M. Lenzerini. Structured objects: Modeling and reasoning. In *Proc. of DOOD-95*, volume 1013 of *LNCS*, pages 229–246, Berlin, 1995. Springer.
2. K. Currie and A. Tate. O-Plan: The Open Planning Architecture. *AI*, 52(1):46–86, 1991.
3. K. Erol. *Hierarchical Task Network Planning: Formalization, Analysis, and Implementation.* PhD thesis, The University of Maryland, 1995.
4. K. Erol, J. Hendler, and D. S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In *Proc. of AIPS-94*, pages 88–96, Chicago, IL, 1994. American Assoc. for AI, AAAI Press, Menlo Park, California.
5. T. A. Estlin, S. A. Chien, and X. Wang. An argument for a hybrid HTN/operator-based approach to planning. In *Proc. of ECP-97*, volume 1348 of *Lecture Notes in AI*, pages 182–194, Berlin, Sept. 24–26 1997. Springer.
6. U. Fonda, A. Natali, and A. Omicini. An object-oriented approach to planning. In *FAPR'96 Workshop "Reasoning about Actions and Planning in Complex Environments"*, pages III–1/8, Darmstadt, Germany, 1996.
7. F. Giunchiglia. Using ABSTRIPS abstractions – where do we stand? IRST Technical Report 9607-10, Istituto Trentino di Cultura, Italy, Jan. 1997.
8. S. Kambhampati, A. D. Mali, and B. Srivastava. Hybrid planning for partially hierarchical domains. In *Proc. of AAAI-98*, pages 882–888, 1998.
9. C. A. Knoblock. Automatically Generating Abstractions for Planning. *AI*, 68:243–302, 1994.
10. A. Mali and S. Kambhampati. Encoding HTN Planning in Propositional Logic. In *Proc. of AIPS-98.* AAAI Press, 1998.
11. D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proc. of AAAI-91*, pages 634–639, 1991.
12. T. McCluskey. Object transition sequences: A new form of abstraction for HTN planners. In *Proc. of AIPS-2000*, pages 216–225, 2000.
13. T. McCluskey, D. Kitchin, and J. Porteous. Object-centred planning: Lifting classical planning from the literal level to the object level. In *Proc. of 11th IEEE Int. Conf. on Tools with AI*, Toulouse, 1996. IEEE Press.
14. D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proc. of IJCAI-99*, pages 968–975, S.F., 1999. Morgan Kaufmann Publishers.
15. D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. SHOP and M-SHOP: Planning with ordered task decomposition. Technical Report CS TR 4157, University of Maryland, 2000.
16. S. Russell and P. Norvig. *Artifical Intelligence: A Modern Approach,.* Prentice-Hall, Englewood Cliffs, NJ, ISBN 0-13-103805-2, 912 pp., 1995, first edition, 1995.
17. E. D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *AI*, 5:115–135, 1974.
18. A. Sernadas, C. Sernadas, and J. F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.
19. W. Stephan and S. Biundo. A New Logical Framework for Deductive Planning. In R. Bajcsy, editor, *Proc. of IJCAI-93*, pages 32–38. Morgan Kaufmann, 1993.
20. W. Stephan and S. Biundo. Deduction-Based Refinement Planning. In B. Drabble, editor, *Proc. of AIPS-96*, pages 213–220. AAAI Press, 1996.
21. A. Tate, B. Drabble, and J. Dalton. The Use of Condition Types to Restrict Search in an AI Planner. In *Proc. of AAAI-94*, pages 1129–1134. AAAI Press, 1994.
22. Q. Yang. *Intelligent Planning: A Decomposition and Abstraction Based Approach to Classical Planning.* Springer, Berlin, 1997.
23. Q. Yang, J. Tenenberg, and S. Woods. On the implementation and evaluation of abtweak. *Computational Intelligence Journal*, 12(2):295–318, 1996.
24. R. M. Young, M. E. Pollack, and J. D. Moore. Decomposition and causality in partial order planning. In *Proc. of AIPS-94.* American Assoc. for AI, AAAI Press, 1994.

On the Adequacy of Hierarchical Planning Characteristics for Real-World Problem Solving*

L. Castillo, J. Fdez-Olivares, A. González

Departamento de Ciencias de la Computación e Inteligencia Artificial
E.T.S. Ingeniería Informática. Universidad de Granada
18071 Granada. {L.Castillo,faro,A.Gonzalez}@decsai.ugr.es

Abstract Starting from a set of requirements which should be accomplished by any hierachical planning approach to real-world problem solving, we show how known hierarchical models may be improved by means of a hybrid approach. Additionally, on the basis of this hybrid model, which is being applied to a real domain, we present a compared criticism about known hierarchical planning properties and its usefulness to reflect the complexity of real-world problems.

Keywords: hierarchical planning, abstraction formalism, planning in real domains

1 Introduction

The use of hierarchical planning techniques is intended to achieve two goals. On the one hand, to improve the efficiency in time and space of non-hierarchical planning methods [3,4,14,17,18,24,23]. And, on the other hand, to achieve problem solving strategies closer to those exhibited by humans in real-world problems [5,10,11,20]. In any case, these hierarchical planning techniques search for a sequence of plans $\{\mathcal{S}^1,..,\mathcal{S}^n\}$ such that $\mathcal{S}^1$ is a solution at the highest level of abstraction, $\mathcal{S}^n$ is a solution at the lowest level of abstraction, that is, a primitive solution, and every plan $\mathcal{S}^{i+1}$ is a valid refinement of a more abstract plan $\mathcal{S}^i$.

In order to achieve those goals, a hierarchical planning model for real-world problems should be based both on some knowledge abstraction formalism for domains descriptions and also on some planning process able to obtain abstract plans and refine them into primitive plans, but in any case, these planning models should satisfy the following requirements:

- Expressiveness. The domain description language and the abstraction formalism should support the representation of real-world problems similarly to hierarchical problem solving knowledge representations of humans.
- Simplicity. The stage of domain description should be as simple as possible, that is, it must have the lowest number of syntactical restrictions and, if possible, it must be supported by knowledge acquisition tools.
- Autonomy. A hierarchical planning model should reduce at maximum the number of decisions taken during domain description, most of them manually defined by humans, and translate these decisions into the planning process which reaches a higher responsibility in the problem solving process. The fact is that domain descriptions may be used to code some procedural knowledge, that in many cases may be an excessive amount of this type of knowledge. The goodness of this practice is not clear. On the one hand, the encoding of procedural knowledge in domain descriptions may be very efficient and may be a straightforward domain description technique. However this could lead the planner not to solve some instances of problems if some "precoded knowledge" is missing. On the other hand, a more declarative knowledge representation in domain descriptions may be more difficult or not very intuitive to handle but, because of its generality, the scope of solvable instances of problems is increased.
- Soundness and completeness. A hierarchical planning process should find valid solutions at every abstraction level whenever they exist.
- Efficiency. In order to adequately exploit a hierarchy of knowledge, a hierarchical planning process should intensively reuse the knowledge embedded in higher level solutions as a guide to refine lower level solutions more efficiently.

* This work has been supported by the spanish government CICYT under project TAP99-0535-C02-01.

Known models of hierarchical planning hardly satisfy all of these requirements, but only some of them, since there seems to be a trade-off between the attainment of ones with respect to others. In general, this is true both in non-decompositional models based on abstraction hierarchies [3,17,18,24] and in decompositional models, either HTN-based models [11,16,20] or POCL-decomposition models [12,13,25].

In the case of decompositional methods, mainly HTN methods, it is widely recognized that they have a great expressiveness power for real world problems [16], however, there is a high number of decisions which must be taken during the description stage of a domain, decreasing the autonomy of the planning process and increasing the effort needed to represent a domain [20,23]. In the case of non-decompositional methods, mainly abstraction hierarchies, they are almost completely devoted to improve the efficiency of planning processes, disregarding expressiveness issues. They represent an excellent theoretical framework for the study of hierarchical planning processes, but its lack of expressiveness makes these methods more difficult to apply in real-world problems. These models may lead to conclude that any improvement in expressiveness implies a decrease in efficiency, and this is not always true [8].

The need to satisfy the above mentioned requirements has lead to the definition of a set of properties which only appear in hierarchical planning models. The main properties which can be found in the literature are *Upward Solution Property*, *Monotonic*, *Ordered Monotonic* and *Downward Refinement Property* [3,17,23].

These properties are intended to define a reference framework for the design of "good" hierarchical planning models so that any model which meet these properties, also satisfies some of the requirements, mainly, efficicency and completeness. However, despite its undoubted theoretical usefulness and the benefit achieved in the development of hierarchical planning models, this set of properties may be questioned because they do not take into account all of the requirements and some property may put in serious risk some of the requirements, mainly expresiveness, i.e. an appropriate representation of knowledge, one of the most important issues for solving real-world problems [9].

However, it is possible to achieve a higher attainment of these requirements by means of a hybrid planning process which could use decomposition techniques jointly with operator-based techniques to benefit of the advantages of both models [12]. This work analyzes the main shortcomings of known hierarchical planning methods and presents a hybrid hierarchical planning model which provides a higher accomplishment of the above requirements and that has been used to solve real-world problems [7,8]. The hybrid process presented in [12] is only focused on representational issues, by reusing concepts from HTN and operator-based planning, neglecting details about its impact on the planning process. This work is a deep effort to explicit the syntax and semantics of decomposition and modularity based on a representational scheme which differs from that of HTN and operator-based planning, and also to explicit the impact of this knowledge representation in the planning process.

In order to correctly argue this criticism, the main issues during the design of a hierarchical planning model have been identified, and they will be used to show how known hierarchical planning methods present some shortcomings in every category and how the approach presented in this work solves some of them providing a higher degree of accomplishement of the requirements. These issues are the following ones:

- Issues related to domain description.
  1. The definition of the abstraction formalism.
  2. A syntactic description to articulate the decomposition of actions.
  3. A semantic description to identify valid decompositions and modularity relations.
- Issues related to the planning process.
  1. Completeness issues in backtracking between different abstraction levels.
  2. Consistency issues of the causal structure of plans between different abstraction levels.

Next section will show the shortcomings of known approaches to hierarchical planning under the point o view of these issues, and this same point of view will be used to show the contributions of this paper in Section 3. Finally, some reflections about the role of these issues, and more complex interactions between the requirements are shown.

2 Shortcomings of known hierarchical models

2.1 Abstraction formalisms

Every hierarchical planning model establishes a set of syntactical tools to allow for the description of different abstraction levels in a domain. This abstraction formalism is used during the planning process in a "top-down"

refinement of high level plans into lower level plans which ends when primitive plans are obtained. The definition of this abstraction formalism directly has effects on requirements such as expressiveness, simplicity and autonomy.

An optimal abstraction formalism should allow for a real knowledge abstraction at different levels [5,15,21] so that actions and literals are represented at different granularity levels and the number of syntactical rules and human decisions are reduced to the minimum in the stage of domain description.

Abstraction hierarchies based models use "literal-oriented" abstraction formalisms in which every literal from level i remains at level $i+1$. This is a very hard syntactic restriction since it limits the abstraction of knowledge, mainly actions, which maintain their semantics at every abstraction level, but described with a different number of literals, so that this formalism is unable to represent compound actions. For this reason, it is very difficult to apply in real-world problems [5,21].

On the other hand, decompositional models use an "action-oriented" abstraction formalism, based either on static decompositions (reduction schemes) [11,23] or on dynamic decompositions (decompositon schemes) [12,13,25] which allow for a real abstraction of actions so that lower level actions are represented with a higher granularity that higher level actions. This formalism provides a great expressiveness power for complex problems [16]. However, in these models, every precondition and effect of a compound action must be somehow distributed amongst its constituent subactions [23,25]. This restriction limits the real abstraction of knowledge, since all of the abstraction levels share the same set of literals, it decreases the simplicity of domain descriptions and needs a great effort of human decisions.

2.2 Decomposition mechanism

This is a set of syntactic tools to decompose high level actions into lower level subactions and it influences expressiveness, simplicity, autonomy and completeness. With respect to expressiveness and completeness, the decomposition mechanism must allow for alternative decompositions and a true modularity of actions, that is, a different granularity of knowledge between the compound action and its subactions. With respect to simplicity and autonomy, action decomposition should provide the means to describe domains with the minimum amount of knowledge supplied by humans.

Action decompositions are present in decompositional models but do not appear in abstraction hierarchies. Although there are some minor differences between them, in both models the decomposition mechanism always need extra knowledge which has to be coded by hand. It is based on a set of static reduction rules which must be supplied by hand during domain description: the antecedent of the rule is a compound action and its consequent is the set of subactions, its relative ordering a set of causal links between them and even binding constraints. These mechanisms are very expressive but also make the stage of domain description very difficult.

2.3 Semantic validity of decompositions

The decomposition mechanism establishes a modularity relation between two plans $\mathcal{S}^i$ and $\mathcal{S}^{i+1}$ at different abstraction levels so that every action in $\mathcal{S}^i$ is mapped into a set of actions of $\mathcal{S}^{i+1}$. Not every syntactically valid decomposition is also a semantically valid one, so a set of criteria which feature semantically valid decompositions, i.e. modularity relations, should be defined taking into account the following issues.

- Every action at level $i+1$ must be related to an action at level i.
- A decomposition of an action should not have any internal unsolvable flaw.
- Subactions should not interfere with the effects of higher level of the action whose decomposition they belong to.

In known decompositional models these issues are responsibility of the human who writes a domain description, decreasing the autonomy of the planner. This shift in responsibility could be avoided by defining general semantic properties which must be satisfied by any modular decomposition, and giving more responsibility to the planner to find a valid decomposition by means the syntactic tools of the mechanism and reducing the effort of coding a domain.

2.4 Backtracking between abstraction levels

A key issue in any hierarchical planning algorithm is the need of backtracking between different abstraction levels during the search process. This is particularly important when at some abstraction level i a solution cannot be found and the algorithm needs to go back at level $i-1$ to search for other refinements. This can be seen as a knowledge-based pruning mechanism which is able to reject dead-end branches at high abstraction levels and provide an improvement in efficiency. However, depending on the abstraction formalism and on the features of a valid solution, there may be some *domains* in which a planner may lose its completeness due to this backtracking mechanism [17,23].

The existence of these domains led to the definition of the *Upward Solution Property* (USP) [17,23]. This property states that if a primitive solution exists $\mathcal{S}^n$ in a hierarchical domain, then there is a sequence of refinements $\{\mathcal{S}^1,..,\mathcal{S}^n\}$ which starts at the highest abstraction level and ends at the lowest level such that every $\mathcal{S}^{i+1}$ is a valid refinement of $\mathcal{S}^i$. It may be seen that the contrapositve of this property allows for backtracking when a solution does not exist at any abstraction level, i.e., there will be no solution at primitive level.

This property is always satisfied by abstraction hierarchies based models [3,17]. But in the case of HTN based models, it is shown [23] that this property does not usually hold but it can be satisfied by the addition of some syntactic restrictions, mainly the *Unique Main Subaction* (UMS). This restriction states that all of the preconditions and effects of a compound action must be reproduced in only one subaction of its decomposition. This restriction leads to use the same granularity of knowledge in compound actions as well as in their subactions, thus completely losing the modular relation of a real decomposition of actions. This implies a lose in expressiveness in real-world problems and, therefore, most HTN based planners do not satisfy the USP. One might think that if USP is not satisfied, then completeness of decompositional planning algorithms may be put in risk in those situations in which the algorithm bactracks but, actually, it depends on the representation of the final solution.

When the solution to a problem is seen as the lowest level plan, i.e., a primitive plan, then hierarchical planning methods may be seen as another heuristic to improve the efficiency of planning, that is, a different way to arrive at a one-level plan which solves a planning problem. In these cases completeness may be lost since in some domains, there can be a primitive solution but no abstract solutions and thus a backtracking criterium based on the nonexistence of abstract solutions would not be enough. On the opposite, when a solution to a problem needs a complete hierarchy of valid plans where every abstract plan is completely autonomous and operational, and it is used as a modularization tool for lower level plans, then the nonexistence of any abstract solution impedes the existence of a complete hierarchy of valid plans and it can be used as a sound backtracking criterium.

Another property related to USP is the *Downward Refinement Property* (DRP) [2,3]. The DRP states that if a non-abstract, concrete level solution to the planning problem exists, then any abstract solution can be refined to a concrete solution without backtracking across abstraction levels. This property is also hard to satisfy in real-world problems. Its fulfilment implies a drastic improvement in efficiency since a hierarchical planner does not need to backtrack between abstraction levels, but in order to do that, there cannot be more than one decomposition for every compound action. This is also very restrictive for real-world problems in which the need to represent alternative decompositions has been widely recognized as a key requirement [16,20].

2.5 Consistency of causal structures through abstraction levels

This last issue consists of reusing the set of causal links established at some abstraction level i as a guide to refine a plan at level $i+1$. When every causal link established at level i is somehow "inherited" at level $i+1$ then a hierarchical planner is said to satisfy the *Monotonic Property* [2,17,23,24]. This property is very important with respect to the attainment of efficiency and consistency of the planning process:

- The efficiency may be improved since the reuse of causal links which have been established at higher levels avoids the redundancy of the planning process in the sense that flaws previously solved at higher levels do not need to be reconsidered again at every lower level. Furthermore, unsolvable threats may be detected earlier to prune dead-end branches.
- From the point of view of the consistency of the planning process, it would not be easily understandable that any action in a plan at some level could contradict any causal relation previously established at any previous level despite of the correctness of that plan in its own level of abstraction.

Therefore, a hierarchical planning model for real-world problems should satisfy this semantic property. In the case of abstraction hierarchies, this property directly holds, due to the abstraction formalism and independently of

the planning process followed, either ABSTRIPS [3,17] or ABTWEAK [24]. HTN based models provide the means to inherit causal links between abstraction levels, but once again, they must be hand-coded during domain descriptions [23] decreasing the autonomy of the planner, whereas in other decompositional models this issue is not addressed [12,13,25].

Next section will show the main contribution of this paper and how it provides helpful advances in every issue discussed in this section with respect to both non decompositional and decompositional approaches to planning for real-world problems.

3 A hybrid model for real-world problem solving

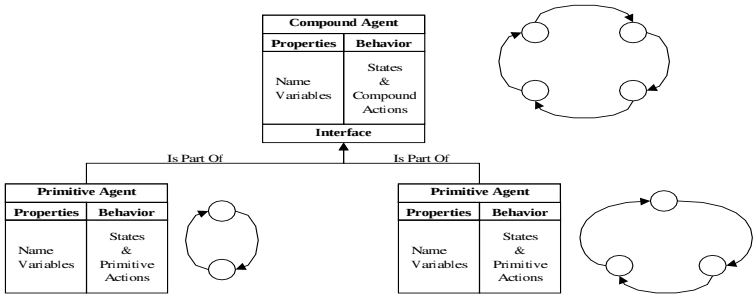


Figure1. Primitive and Compound agents.

In our model a domain is represented as a *compositional hierarchy of agents* (see Figure 1), i.e., an agent hierarchy with different levels of abstraction such that high-level agents (*compound agents*) are composed by lower-level agents. Leaf nodes of the hierarchy are *primitive agents*, which represent real world entities able to act. We can find many real applications which fit into these features [22] (robot planning and control [1], manufacturing systems [6,19] or aerospace applications [12]).

Every agent g of a compositional hierarchy is represented by means of its properties (name and variables) and its behaviour. The behaviour is modeled as a finite automaton in which every action a of g is represented by a set of requirements, *Req*(a), which must be satisfied in order to achieve a correct execution of the action, and a set of effects, *Efs*(a), which include the change of state produced by a over the agent g (we note *Ag*(a) the agent which an action a belongs to). In order to achieve an adequate expressiveness for real-world planning, the model of actions embodies two fundamental features:

- Actions are considered as intervals, that is, every action a of an agent g executes over an interval, $[a, \mathit{End}(a)]$, defined from a until the next change of state of g, produced by another action of the same agent, *End*(a).
- The set of requirements is divided into a set of condition types to represent different ways for preconditioning an action (see [7,8] for more details).

Actions in a compositional hierarchy may be *primitive actions*, if they are executed by primitive agents, or *compound actions*, if they are executed by compound agents. Both action types share the same structure, but compound actions, which are represented at a higher level of abstraction, additionally include a set of *expansion methods*, which are used to decompose them into subactions. Details about this decomposition mechanism will be described later in Section 3.2.

Remaining sections are devoted to explain in more detail our hybrid model, and to show how the key issues of known hierarchical models, shown in Section 2, may be improved in order to solve real-world problems.

3.1 Abstraction formalism

As opposite to the formalisms described in Section 2.1, the abstraction formalism of our hybrid approach is *behaviour oriented*, that is, the abstraction is centered on the behaviour of compound agents in such a way that the behaviour of high level agents is a more abstract representation of the behaviour of their components.

Firstly, the knowledge of a compound agent and the knowledge of its components are related by means of the *Interface* of the compound agent [6] (noted as Int_g). This is a set of simple association rules used to map literals, variables and states of an agent into literals, variables and states of its components. This mechanism allows an expert to easily describe and relate actions and literals of agents at different levels with a different semantic granularity, such that sets of literals and actions at different abstraction levels are also different.

Secondly, in order to articulate levels of abstraction downwards, during the planning process, we have defined an *articulation function* [6,15]. This function (noted as f_g) is a domain independent mechanism defined for every compound agent g which uses its interface to translate every literal into a new level of abstraction. Hence, every literal l=(N $x_1 \ldots x_n$) at level i, in the description of a compound action α^1 of a compound agent g, is translated into a set of literals of level $i+1$, whose arguments are consistent with the new level of abstraction. These literals are built according to these rules:

- If there is a rule in Int_g which specifically associates l to a set of literals, then $f_g(l)$ returns that set.
- Otherwise, the literal is translated component by component, i.e., $f_g(l)$ returns the single set $\{$ ($Int_g(n)$ $Int_g(x_1) \ldots Int_g(x_n)$)$\}$
- Finally, if there is no semantic correspondence in the next abstraction level for l, $f_g(l)$ returns the empty set.

Both the articulation function and the interface of compound agents may be used to relate knowledge at different granularities. The knowledge of a compound agent, with a lower granularity, may be translated into the knowledge of its constituent agents, with a greater granularity.

Additionally, there is a sort of assisting tool for domain descriptions: an expert may describe a domain as a compositional hierarchy from an existing *class library* of predefined agents which also contains predefined agent interfaces. That is, rules contained in interfaces may be easily reused, simplifying the process of domain encoding.

This abstraction mechanism provides a real abstraction of knowledge since actions and literals in the hierarchy are represented at different granularity levels, but related between them. As a result, the amount of syntactic constraints which must be observed during the domain description stage is reduced. There is no need to distribute preconditions and effects at lower abstraction levels since this knowledge will be translated by the articulation function. Moreover, a human domain designer only has to specify which are the components of a compound agent and, if needed, to specialize some agents by introducing domain specific knowledge.

As will be seen later, this formalism is able to maintain an HTN-like expressiveness, but with a simpler domain description process which reduces human decision making at early stages.

3.2 Decomposition Mechanism

Every compound action α of every agent at level i, contains a set of *expansion methods* $\{m_0, m_1, \ldots\}$ where every method $m_{j,j>0}$ is a set of literals at level $i+1$ which represent subgoals to be achieved by actions of agents at the next level $i+1$. This set of literals will be noted as *Decomp*(α) and they are a semantically equivalent representation of the effects of α at level $i+1$.

From the point of view of a human at the domain description stage, it must be taken into account that every compound action α contains a *default expansion method*, m_0, whose literals are the result of applying f_g to every literal in the set of effects of α. Moreover, in most cases this method has shown to be expressive enough to describe how a compound action may be decomposed.

The decomposition mechanism is a dynamic process, performed at planning time, which decomposes a compound action α at level i following these two steps:

1. Determine the set of literals at the next level $i+1$ which have to be achieved by actions of agents at that level. It may be obtained from m_0, without any additional knowledge, or by means of another alternative method. .
2. Determine, by means of POP-based techniques, the set of actions such that either they satisfy those literals generated by α or they contribute to their establishment. These actions will make up the real decomposition of α.

This decomposition process preserves the expressiveness of decomposition rules of other models, that is, it allows for alternative action decompositions, with different grain size. Additionally, it improves decomposition mechanisms discussed in Section 2.2 in the sense that it does not require to know the modular decomposition

[1] Greek letters represent higher level actions and latin letters lower level actions.

of every compound action, prior to the planning process, and in most cases the only required knowledge is the interface of every compound agent, covering much better requirements of simplicity and autonomy.

Another feature of this mechanism is that the correctness of a decomposition does not have to be checked "a priori" by hand. Instead of this, this task is shifted into the planning process who becomes the responsible to dynamically check the correctness of every decomposition at every level of abstraction.

3.3 Semantic validity of decompositions

A key issue in the previously described decomposition process is that it allows to shift the task of determining the set of subactions which corresponds to a compound action into the planner, while preserving semantic correctness. This feature leads to define a *modularization relationship* between two plans at different levels of abstraction. In the following we will introduce some fundamental concepts that, finally, will allow us to define what we consider a valid decomposition as a *correct modularization relationship*.

Modularization relationships In our compositional hierarchy we use two functions, *Sub*(α) and *Scope*(a), to represent hierarchical relationships "is composed by" and "is part of" respectively, so we say that $a \in$ *Sub*$(\alpha) \Leftrightarrow \alpha =$ *Scope*(a). Function *Scope*(a) (Figure 2) gathers the general criteria that a hierarchical planer has to take into account to dynamically find the set of subactions $\{a_1, a_2 \ldots\}$ of a compound action α. It takes as input an action a, from a plan $\mathcal{S}^i$ at level i, which satisfies a literal l from the same level, and it decides which is the *scope* of a, that is, the compound action α at the inmediately higher level which contains the action a in its decomposition.

Scope(a)
IF $\exists \alpha$ / $a \xrightarrow{Sat} l, l \in$ *Decomp*(α)
THEN Let $\{\alpha_1, \ldots, \alpha_n\}$ such that $a \xrightarrow{Sat} l_i, l_i \in$ *Decomp*(α_i)
Let α = Non Deterministically Choose One Of **First**$\{\alpha_1, \ldots, \alpha_n\}$
ELSE Let $\{a_1, \ldots, a_n\}$ such that $a \xrightarrow{Sat} l_i, l_i \in$ *Req*(a_i)
Let α = Non Deterministically Choose One Of **First**$\{$*Scope*$(a_1), \ldots,$ *Scope*$(a_n)\}$
Return α

Figure2. Function *Scope*

Summarizing, the scope of an action a is a higher level action α if one of the following conditions holds:

- a establishes several literals generated by different higher level actions, $\{\alpha_1, \ldots, \alpha_n\}$, and α is one of the first actions of this set (function **First** returns a set of actions for which no other actions precede them in the set of actions used as argument). It must be said that a same action may contribute to different higher level actions but it is assigned to only one of them.
- a establishes several requirements of a set of actions at its same level, $\{a_1, \ldots, a_n\}$, and α is one of the first *scopes* of this set of actions.

This definition is based on literal satisfaction, so it is possible to dynamically generate, at planning time by means of POP-based techniques, the set of subactions for a given compound action at level i. Moreover, functions *Scope*(a) and *Sub*(α) establish a modularization relationship between plans at two consecutive abstraction levels, such that every action in a plan at level i is mapped into a set of actions of a plan at the next level $i+1$.

Taking into account the terms of this definition, POP-based causal link management is the natural way to dynamically guarantee a valid causality between the subactions of a compound action.

Correct modularization relationships In order to guarantee a correct modularization relationship, high-level effects of a compound action α at level i should not be deleted by any of its subactions at level $i+1$. The set of literals *Decomp*(α), generated at level $i+1$ by the above described decomposition mechanism of α, are a semantically equivalent representation of the effects of α at level i, therefore these are actually the literals which should be protected at level $i+1$.

The new concept of *hybrid causal link* is used to protect these literals. In order to understand how a hybrid causal link is represented, and its semantic implications, it is necessary to know the extended representation of a causal link used in our model, which embodies the notion of actions interval, a more suitable representation for real-world domains [7,8].

A causal link $[a \xrightarrow{l} b, c]$ is a structure used for representing that a requirement l of an action b has been satisfied by another action a and this has to be protected during the action interval $[a, c]$ and c is not necessarily $End(a)$. So, an action a' threatens a causal link $[a \xrightarrow{l} b, c]$, when a' deletes l and the interval $[a, c]$ is unordered with respect to the interval $[a', End(a')]$[2]

Definition 1 (Hybrid causal link.) *A hybrid causal link at level i is a structure represented as $[a_\alpha \xrightarrow{l} \beta, \gamma]$, where*

- *β and γ are two compound actions at level $i-1$ which belong to a causal link $[\alpha \xrightarrow{r} \beta, \gamma]$ at level $i-1$.*
- *l is a literal at level i generated by α, at level $i-1$, such that $l \in f_{Ag(\alpha)}(r)$.*
- *a_α is an action which satisfies l and $\alpha = Scope(a)$.* □

A hybrid causal link describes that a literal generated by α and satisfied by some subaction a_α of α has to be protected from any other threatening action a', at the same level than a_α, which could delete that literal. Thus, a hybrid causal link will be used to detect and solve a *hybrid threat*, an analogous concept to that of classic threat which takes into account the existence of causal links at different abstraction levels.

Definition 2 (Hybrid Threat.) *An action a' produces a hybrid threat to a hybrid causal link $[a_\alpha \xrightarrow{l} \beta, \gamma]$ when*

i) $a' \xrightarrow{Del} l$
ii) $[a', End(a')]$ *is unordered with respect to* $[a, \gamma]$ □

Finally, based on these concepts of hybrid causal link and hybrid threats we are able to establish what we consider a *correct modularization relationship*. A correct modularization between two plans at consecutive abstraction levels exists when there are no hybrid threats within the scope of any compound action, that is, when any effect of a compound action is not deleted by any of its subactions.

Therefore, any modularization relationship must satisfy that

$$\nexists a_\alpha, a'_\alpha \in Sub(\alpha), \text{ such that } a'_\alpha \text{ produces a hybrid threat to a hybrid causal link } [a_\alpha \xrightarrow{l} \beta, \gamma].$$

The definition of a correct modularization relationship is a particular case of a hybrid threat in which the threatening action and the causal link belong to the same scope. It allows a planner to decide which decompositions are correct and also to establish the order relations and causal links between its subactions, both dynamically. Moreover, from the point of view of a human domain designer, the planner will be able to modularize the actions of one level with respect to the actions of a higher level, a task which is only carried out by humans in HTN methods.

This procedure to dynamically identify valid decompositions of actions allows for an increase of autonomy of the planner and a greater simplicity of domains descriptions, since the knowledge which identifies these valid decompositions does not have to be provided during domain coding. Instead, this knowledge is "distilled" during the planning process.

3.4 The planning process

The goal of the planning process is obtaining a hierarchical plan which correctly describes the behaviour of a compositional hierarchy of agents at different levels of abstraction. A solution to a problem is a hierarchical plan, that is, a sequence of plans $\{\mathcal{S}^1 \ \ldots \ \mathcal{S}^n\}$ at different abstraction levels, where every plan $\mathcal{S}^i$ must be a valid solution at level i, which will be called a level i partial solucion, since it describes the behaviour of agents at level i. The lowest level plan is completely made up of primitive actions, and every plan at level i has a correct modularization relationship with respect to the plan at the next abstraction level $i+1$. The algorithm which generates such hierarchical and modular plans is shown in Figure 3.

[2] Two intervals are unordered when their limits are unordered [8].

HYBIS(Domain, Agenda, H-Plan)
IF Agenda *is empty* AND **IsPrimitivePlan**(H-Plan[CurrentLevel])
THEN RETURN H-Plan
ELSE LET Flaw = **SelectFlaw**(Agenda)
IF Flaw *consists on a hierarchical refinement*
THEN RETURN **REFINE**(Domain, Flaw, Agenda, H-Plan)
ELSE RETURN **GENERATE**(Domain, Flaw, Agenda, H-Plan)

REFINE(Domain, α, Agenda, H-Plan)
LET Methods = **HowToRefine?**(α, Domain, H-Plan)
WHILE Methods *is not empty*
LET m = **ExtractExpansionMethod**(Methods)
Insert(m, Plan-H])
LET Result = **HYBIS**(Domain, Agenda, H-Plan)
IF Result $\neq$ FAIL THEN RETURN Result
RETURN FAIL

GENERATE(Domain, Flaw, Agenda, H-Plan)
LET Alternatives = **HowToDoIt?**(Flaw, Domain, H-Plan)
WHILE Alternatives *is not empty*
LET How = **Extract**(Alternatives)
DoIt(How, Domain, Agenda, H-Plan)
LET Result = **HYBIS**(Domain, Agenda, H-Plan)
IF Result $\neq$ FAIL THEN RETURN Result
RETURN FAIL

Figure3. A hierarchical hybrid algorithm based on hierarchical HTN-like refinement and generative POP-like refinement.

This algorithm synthesizes a hierarchical plan by generating and refining a set of plans at different levels of abstraction. Every plan $\mathcal{S}^i$ describes the behaviour of a set of agents at the same level, and it may be considered a complete solution to a level i problem or a partial solution to the whole problem.

The highest level plan, $\mathcal{S}^1$, is obtained by generative techniques, and every plan $\mathcal{S}^i$, for $i > 1$, is obtained by means of a hybrid process which interleaves hierarchical (function **REFINE**) and generative (function **GENERATE**) refinements. Function **REFINE** performs the step 1 described in Section 3.2, that is, it decomposes every compound action α into a set of literals by any of its existing methods and then it inserts these literals in the plan. Function **GENERATE** performs the step 2 described in Section 3.2 taking into account the validity of decompositions described in Section 3.3. That is, it satisfies every literal by means of generative techniques and establishes a correct modularization relationship between every compound action and its subactions. Function **GENERATE** is based on a non-hierarchical POCL planner described in [8].

The process ends when all of the actions in the current plan are primitive actions and that plan has been correctly modularized.

Next, we will discuss the key issues of backtracking between levels of abstraction, and the consistency of causal structure through levels of abstraction.

3.5 Backtracking between abstraction levels.

In the algorithm described in Figure 3, backtracking between abstraction levels is done when there is no possibility to generate a plan $\mathcal{S}^i$ modular and causally correct with respect to $\mathcal{S}^{i-1}$ as seen in Section 3.3. It must be said that, taking into account the type of solutions obtained by our approach, backtracking between abstraction levels has no negative effect over the completeness of the algorithm, as discussed in Section 2.4.

A solution is not a primitive plan, as in most HTN approaches, but a complete hierarchical and modular plan, that is a sequence of plans $\{\mathcal{S}^1 \ \ldots \ \mathcal{S}^n\}$, where the lowest level plan is completely made up of primitive actions, every plan $\mathcal{S}^i$ is a partial solution at level i and it has a correct modularization relationship with respect to the plan at the next abstraction level $i + 1$. Hence, if there is no partial solution at some abstraction level, then there is no possibility to obtain such a complete hierarchy of valid plans, despite of the existence of any later primitive partial solution. Therefore, in these situations, the need to obtain a complete hierarchy of plans leads to backtrack to the previous abstraction level to continue the search for alternative refinements. Given that a solution is a complete hierarchy of plans, as opossite to most HTN approaches in which a solution is a primitve plan, a backtracking criterium based on the contrapositve of USP does not put in risk the completeness of the algorithm.

As can be seen, the usefulness of the USP to backtrack between abstraction levels in real-world planning depends on how a correct solution is defined. However, our experience in solving real-world planning cases points to an interesting role of this property in domain validation, which will be outlined in Section 4.

Finally, next section will be devoted to show the monotonicity of the causal structure through different abstraction levels.

3.6 Consistency of causal structures through abstraction levels.

In our hybrid model, a plan $\mathcal{S}^i$ inherits the causal link structure generated in the plan $\mathcal{S}^{i-1}$ by means of a dynamic process at planning time. This process is based on reusing the high level structure of causal links, taking into account the inheritance rules shown in Figure 4.

RULE 1:	**RULE 2:**
A causal link $[\alpha \xrightarrow{r} \beta, \gamma]$ from $\mathcal{S}^{i-1}$, such that $f_{Ag_{(\alpha)}}(r) \neq \text{NIL}$, has associated a hybrid causal link $[a_\alpha \xrightarrow{l} \beta, \gamma]$ in $\mathcal{S}^i$ if i) $l \in f_{Ag_{(\alpha)}}(r)$ ii) $a_\alpha \xrightarrow{Sat} l$	A hybrid causal link $[a_\alpha \xrightarrow{l} \beta, \gamma]$ from $\mathcal{S}^i$ has associated a causal link $[a \xrightarrow{l} b, c]$ in $\mathcal{S}^i$ if i) $b \in Sub(\beta)$ ii) $a \xrightarrow{Sat} l$, $l \in Req(b)$

Figure4. Inheritance rules of causal links

This monotonic inheritance of higher-level causal structures is used to detect and solve hybrid threats. These hybrid threats represent a violation of a causal link established at level i by an action at level $i+1$ and they will become a classic threat at level $i+1$ when all of the actions at level i had been decomposed. Hence, unsolvable hybrid threats, which will later become unsolvable classic threats, may be used for an early detection of dead-end branches and backtracking before all of the decompositions have been completed.

In summary, this inheritance of causal structures not only provides a greater expressiveness, since it is the basis of the dynamic decomposition mechanism, but it also provides a greater efficiency, since it allows for an early detection of some unsolvable threats exploiting the knowledge synthesized at previous levels.

4 Conclusions

In this work we have shown some advances in hierarchical planning which overwhelm a set of shortcomings on known hierarchical models, and we have presented a discussion on the adequacy of hierarchical planning properties for real-world planning, on the basis of a hybrid planning approach developed to solve real-world problems.

Space precludes to justify our proposal with an appropriate experimentation. It must be said that our model has been extensively used for the automatic synthesis of hierarchical control programs for manufacturing systems in which simple domain descriptions and modularity relations play a very important role. This experimentation is being carried out in close collaboration with experts on industrial domains within a research project, and it will appear in other paper in preparation.

In any case, all of the improvements which have been proposed have a common basis: the abstraction formalism based on the *articulation function*. This abstraction formalism allows for a high accomplishment of formerly enumerated requirements (see Table 1 for more details):

- Simplicity: it reduces human effort on syntactic constraints observance and decision making, at domain description stage.
- Autonomy and expressiveness: it provides the basis for a dynamic action decomposition process performed at planning time and it allows to obtain "ready-to-use" plans which describe the behaviour of a compositional hierarchy of agents.
- Soundness and completeness: it also provides the basis for dynamically checking the semantic correctness through plan levels with different semantic granularity, while preserving completeness.
- Efficiency: the inheritance of causal structures provides the means for an intensive reuse of the knowledge embedded in higher levels. This reuse of higher level knowledge produces a great benefit on the efficiency and, in addition, it allows for a more understandable planning process from the point of view of a human.

On the other hand, we have also discussed the adequacy of known hierarchical planning properties for real-world problem solving. In summary, the DRP is too restrictive and it leads to reduce expressiveness for real world problems. However, we have shown that the Monotonic Property is very useful for real-world planning. In particular, we have introduced a new mechanism which monotonically inherits causal structures between plans at different abstraction levels, which are represented at different semantic granularity by changing the representation

language. Finally, we have shown that USP may be used to bactrack between abstraction levels without putting in risk the completeness of our hybrid algorithm. This is because the type of solution needed in our approach is not a primitive plan, but a complete hierarchy of plans. However, the accomplishment of this property is still useful during domain description process, from a knowledge engineering perspective.

Although the completeness of the planning algorithm could be formally proven, it is always possible to describe a hierarchical domain with no partial solution at some abstract level, but with a partial solution at ground level. This means that the USP is an inherent property to every hierarchical domain, whose accomplishment is necessary to prevent the description of bad domains. However, it could be argued that there could be some feedback between the planning process and the domain description process, such that the planner could be able to detect and suggest some domain coding errors. This strategy could be implemented by means of a mixed initiative planning process, such that the planner would inform the expert about the circumstances which invalidate the execution of actions at a given level (for instance, a unsolvable threat could mean a deadlock between two agents). This situation would interrupt the hierarchical refinement process until a redefinition of conditions at that abstraction level had been done.

This mixed initiative process could provide the basis for a dynamic knowledge validation, at different levels of abstraction, during the planning process. But it must be said that such as mixed initiative redefinition process, at a single level of abstraction, is only operative with an abstraction formalism like the one presented here, since it needs a completely differentiated representation at every abstraction level, otherwise this redefinition should also be propagated into lower level representations.

At present we are developing an intelligent digital assistant for the interactive development of industrial control programs on the basis of this hybrid model.

	Abstraction formalism	**Decomp. mechanism**	**Decomp. validity**	**Backtracking**	**C.Links Inherit.**
Non-Dec. Models	Literal Oriented Abstraction. Same set of literals for every level. Hierarchies may be self-generated.	No	No	Always correct. Satisfy USP,DRP.	Satisfy Monotonic Prop. Direct inheritance.
Decomp. Models	Action Oriented Abstraction. Same set of literals for every level. Syntactic constraints about actions literals.	Reduction schemes. Need extra knowledge. Total or partial description.	It is restricted by hand at domain description.	USP depends on UMS.	Directly inherited or restricted at domain description.
Hybrid Model	Behavior Oriented Abstraction. Granularity levels of knowledge. No syntactic constraints about actions literals.	Default expansion method. No need of extra knowledge.	All the decision taken on planning time. Correct modularization relationship.	Correct, due to solution features. Mixed initiative may help to accomplish USP.	Hybrid c.links are built at planning time. Causal inheritance with change of language.

Table1. Characteristics of hierarchical planinng

References

1. R.C. Arkin. *Behavior-Based Robotics.* MIT press, Cambridge, MA, 1998.
2. F. Bacchus and Q. Yang. The downward refinement property. In *Proceedings of IJCAI 91*, pages 286–292, 1991.
3. F. Bacchus and Q. Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71:43–100, 1994.
4. C. Backstrom and P. Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proc. of IJCAI 95*, pages 1599–1604, 1995.
5. R. Bergman and W. Wilke. Building and refining abstract planning cases by change of representation language. *JAIR*, 3:53–118, 1995.
6. L. Castillo, J. Fdez-Olivares, and A.González. A hybrid hierarchical/operator-based planning approach for the design of control programs. In *Proceedings of ECAI'2000. Workshop on Planning, Scheduling and Design. PUK'2000.*, 2000.

7. L. Castillo, J. Fdez-Olivares, and A. González. A three-level knowledge-based system for the generation of live and safe petri nets for manufacturing systems. *Journal of Intelligent Manufacturing*, 11(6):559–572, 2000.
8. L. Castillo, J. Fdez-Olivares, and A. González. Mixing expresiveness and efficiency in a manufacturing planner. *To appear in Journal of Experimental & Theoretical Artificial Intelligence (JETAI)*, 2001.
9. S. Chien, R. Hill, Jr. X. Wang, and H. Mortenson T. Estlin, K. Fayyad. Why real-world planning is difficult: a tale of two applications. In M. Ghallab and A. Milani, editors, *New directions in AI Plannig*, pages 287–298. IOS Press, 1996.
10. K. Currie and A. Tate. O-Plan: Control in the open planning architecture. In *BCS Expert systems conference*, 1985.
11. K. Erol, J. Hendler, and D. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *AIPS-94*, 1994.
12. T.A. Estlin, S.A. Chien, and X. Wang. An argument for a hybrid HTN/operator based approach to planning. In *Recent Advances in AI Planning.Proc. of 4th European Conference on Planning ECP'97*, pages 182–194, 1997.
13. M. Fox. Natural hierarchical planning using operator decomposition. In *Recent Advances in AI Planning.Proc. of 4th European Conference on Planning ECP'97*, pages 195–207, 1997.
14. F. Giunchiglia. Using abstrips abstractions – where do we stand? *Artificial Intelligence Review*, 13:201–213, 1999.
15. J. Hobbs. Granularity. In *IJCAI 85*, pages 432–435, 1985.
16. C. Knoblock. AI Planning systems in the real world. *IEEE Expert*, pages 4–12, 1996.
17. C. A. Knoblock. *Generating Abstraction Hierarchies*. Kluwer Academic Publishers, 1993.
18. D. E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.
19. S.Viswanathan, C.Johnsson, R.Srinvivasan, V.Venkatasubramanian, and K.E. Arzen. Automating operating procedure synthesis for batch processes. part I: Knowledge representation and planning framework. *Computers and Chemical Engineering*, 22:1673–1685, 1998.
20. D. E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.
21. D. E. Wilkins. *Practical planning: Extending the classical AI planning paradigm*. Morgan Kaufmann, 1988.
22. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, October 1995.
23. Q. Yang. *Intelligent Planning. A decomposition and Abstraction Based Approach*. Springer Verlag, 1997.
24. Q. Yang, J. Tenenberg, and S. Woods. On the implementation and evaluation of ABTWEAK. *Computational Intelligence*, 12:307–330, 1996.
25. R.M. Young, M.E. Pollack, and J.D. Moore. Decomposition and causality in partial order planning. In *Proceedings of the Second International Conference on AIPS*, 1994.

Slack-based Techniques for Robust Schedules

Andrew J. Davenport[1], Christophe Gefflot[2], and J. Christopher Beck[2]

[1] IBM T J Watson Research Center, PO Box 218, Yorktown Heights
New York 10598 USA
davenport@us.ibm.com
[2] ILOG, S.A., 9, rue de Verdun, B.P. 85, F-94523 Gentilly Cedex France
{cgefflot, cbeck}@ilog.fr

Abstract. Many scheduling systems assume a static environment within which a schedule will be executed. The real world is not so stable: machines break down, operations take longer to execute than expected, and orders may be added or canceled. One approach to dealing with such disruptions is to generate robust schedules: schedules that are able to absorb some level of unexpected events without rescheduling. In this paper we investigate three techniques for generating robust schedules based on the insertion of temporal slack. Simulation-based results indicate that the two novel techniques out-perform the existing temporal protection technique both in terms of producing schedules with low simulated tardiness and in producing schedules that better predict the level of simulated tardiness.
Keywords: Robustness, Uncertainty, Scheduling, Heuristics

1 Introduction

Based on a field study of a number of job shops, McKay et al. [MSB88] comment that "the [static job shop] problem definition is so far removed from job-shop reality that perhaps a different name for the research should be considered". In particular, they found that modern scheduling technology failed to adequately address scheduling in uncertain, dynamic environments.

There are two general approaches to dealing with uncertainty in scheduling. Whereas *reactive* techniques address the problem of how to recover from a disruption once it has occurred, *pro-active* scheduling constructs schedules that account for statistical knowledge of uncertainty. One way of achieving this is by generating *robust* schedules, that is, a schedule with "the ability to satisfy performance requirements predictably in an uncertain environment" [LP91].

In this paper, we explore slack-based techniques for robust scheduling. The central idea behind slack-based techniques is to provide each activity with extra time to execute so that some level of uncertainty can be absorbed without rescheduling. We define the amount slack for an activity, A, as follows:

$$slack_A = lft_A - est_A + dur_A \tag{1}$$

Where est_A and lft_A are respectively the earliest start time and latest finish time of activity A and dur_A is the duration of activity A.

Three slack-based techniques are examined in this paper. The first, *temporal protection* [Gao95], adds slack to an activity before scheduling. The original duration of each activity is extended and then this protected duration is used during scheduling. Two novel techniques are introduced here:

- Time window slack (TWS): Rather than extending the durations of activities, this technique modifies the problem definition to ensure that each activity will have at least a specified amount of slack. The advantage of this approach over temporal protection is that the amount of slack for each activity can be reasoned about during the problem solving rather than being "hidden" inside the protected duration.
- Focused time window slack (FTWS): TWS and temporal protection specify the amount of slack required for each activity before problem solving. In FTWS, the amount of slack for each activity depends on where along the temporal horizon an activity is scheduled. The intuition is that the later in the schedule an activity is executed, the more likely it is to have a disruptive event occur before its execution and, therefore, the more slack is needed.

2 Problem Definition

The problem addressed here is the job shop scheduling problem with release and due dates, machine breakdowns, and the optimization criteria of minimization of the sum of job tardiness.

Each job is composed of a set of totally ordered activities. Each job, j, has a release date, $rd(j)$, and a due date, $dd(j)$. The former is the earliest time an activity in the job can execute and the latter is the latest time that the last activity in the job should finish execution. Each activity requires a single resource (also referred to as a machine) and has a predefined duration during which it must be the only activity executing on its required resource. Once an activity has begun execution it cannot be pre-empted by another activity.

The goal is to sequence the activities on each resource such that the order within each job is respected and that the sum of the job tardiness is minimized. More formally, given $C(j)$, the completion time for the last activity in job j, we seek to minimize $\sum \max(0, C(j) - dd(j))$ over all jobs in a problem.

To represent uncertainty, some machines are subject to breakdowns. During a breakdown, a machine cannot process any activities and any activity which was executing on the machine at the time of breakdown is stopped and then resumed from the point where it was stopped after the machine has been repaired. Statistical characteristics of machine breakdowns are known.[1] We assume normal distributions parameterized by:

- $\mu_{tbf}(R)$: the mean time between failure of resource R
- $\sigma_{tbf}(R)$: the standard deviation in time between failure on R
- $\mu_{dt}(R)$: the mean down time (or duration of breakdown) on R
- $\sigma_{dt}(R)$: the standard deviation in down time on R

[1] In a production scheduling domain, such characteristics may be supplied by the manufacturer and/or may be based on shop floor operational history.

3 Temporal Protection

Temporal protection [Gao95] is a preprocessing technique which extends the duration of each activity based on the uncertainty statistics of the resource on which it executes.[2]

Resources that have a non-zero probability of breakdown are designated as *breakable* resources. The durations of activities requiring breakable resources is extended to provide extra time with which to cope with a breakdown. The scheduling problem with protected durations is then solved with standard scheduling techniques.

The intuition behind the extension of durations is that during schedule execution, the protected durations provide slack time which can be used in the event of machine breakdown. For instance, in Figure 1, activities A and B are sequenced to execute consecutively on a breakable resource. The length of the white box represents the original duration of the activities while the shaded box represents the extension due to temporal protection. If the machine breaks down while activity A is executing, the extra time within the protected duration can be used to absorb the breakdown. If the breakdown lasts no longer than the available protection, its effect will not be felt in the rest of the schedule. If the breakdown lasts longer, some reactive approach must be taken to restore consistency to the schedule. If no breakdown occurs during the execution of activity A, then activity B can start earlier: the slack provided by the temporal protection for A is available for use by activity B.

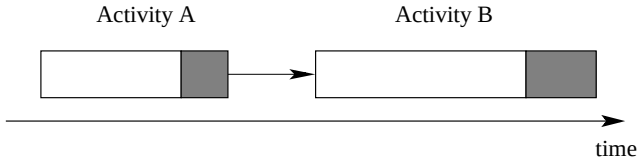


Fig. 1. Example of a temporally protected schedule, with white boxes representing the original duration and grey boxes representing the extended durations.

A key issue is the amount of temporal protection added to each activity. Too much protection will result in a poor quality but highly robust schedule. Too little protection will also result in a poor quality schedule execution if a breakdown occurs. The approach taken by Gao extends the duration so as to amortize the breakdown over a number of activities.

More formally, given an activity, A, that requires a breakable resource, R, temporal protection defines a protected duration for A, $dur_{A,tp}$, where tp denotes temporal protection, as follows:

[2] We present a simplified description of temporal protection in the case that each activity requires only one resource. For a formulation for multiple resource requirements see [Gao95].

$$dur_{A,tp} = dur_A + \frac{dur_A}{\mu_{tbf}(R)} \times \mu_{dt}(R) \quad (2)$$

The equation specifies that the protected duration of an activity A is its original duration plus the duration of breakdowns that are expected to occur during the execution of A.

4 Time Window Slack

By extending activity durations in a preprocessing step, temporal protection transforms the original problem to a new problem that can be solved with any scheduling technique. We conjecture, however, that the preprocessing has a disadvantage in that during scheduling the amount of slack added to each activity cannot be directly reasoned about. This can lead to situations where it is impossible to share slack between activities even when no resource breakdowns have occurred. Indeed, the claim that activity B in Figure 1 can start execution early if there is no breakdown during or before activity A is an over-simplification. For example, in Figure 2 we introduce a third activity, C, which executes on a non-breakable resource and so has no temporal protection. Activity B must execute after activity C, and since activity C finishes execution later than activity A, the earliest start time of B is the end time of C. The temporal protection represented by the extension of the duration of activity A is not available for use by activity B as B cannot start earlier than the end time of C.

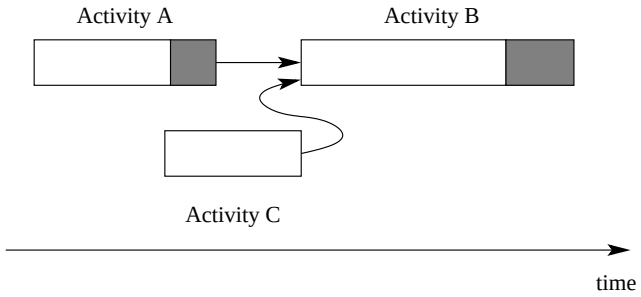


Fig. 2. A situation where the temporal protection cannot be shared between activities A and B.

To avoid such situations, we propose the *time window slack* (TWS) approach which reasons directly about the slack time available for an activity during problem solving. Rather than including this slack as part of the activity duration, we explicitly reason about it by adding a relation to the problem definition that specifies that schedules must have sufficient slack time for each activity.

The advantage of this approach is that there is more information about activity slack during the problem solving. In a situation such as the one in Figure 2,

we are able to detect that the slack of activity A cannot be shared with activity B. If there is still sufficient slack in the schedule after activity B, we still may be able to generate a valid schedule. If not, we must backtrack and continue search.

The amount of slack for each activity is still critical for the generation of a robust schedule. Using $acts_R$ to denote the set of all activities executing on resource R, the required slack for activity $A \in acts_R$ is:

$$slack_A \geq \frac{\sum_{B \in acts_R} dur_B}{\mu_{tbf}(R)} \times \mu_{dt}(R) \tag{3}$$

The required slack for an activity under TWS is considerably larger than the duration extension in temporal protection. Indeed, the amount of slack on each activity is equal to the sum of the durations of all the expected breakdowns on R. This difference is because we expect the slack on all activities on a resource to be shared. If the slack is completely shared then the total slack on a breakable resource is approximately equal (given integer rounding) to the sum of the durations extensions in temporal protection.

The relation in inequality 3 is somewhat problematic for standard scheduling approaches: no solution which assigns start times to activities can satisfy it unless the left-hand side evaluates to 0. The very act of assigning a start time forces the slack (as defined in equation 1) to be 0. Therefore, rather than being able to use arbitrary scheduling techniques, we must use scheduling techniques that reason about the order of activities on resources. Fortunately, such techniques are not uncommon (e.g., [SC93,BF00]).

5 Focused Time Window Slack

Neither temporal protection nor TWS take into account the placement of activities on the scheduling horizon. For example, consider scheduling a newly repaired machine whose $\mu_{tbf}(R)$ is 1000 days and whose $\sigma_{tbf}(R)$ is 50 days. Given the negligible probability of a breakdown before day 800, it does not seem worthwhile to focus on making the schedule more robust before this time. *Focused time window slack* (FTWS) uses the uncertainty statistics to focus the slack on areas of the horizon that are more likely to need it to deal with a breakdown.

The probability distribution, $P(N(\mu_{tbf}(R), \sigma_{tbf}(R)) \leq t)$, allows us to compute the probability that a breakdown event will occur at or before time t. An approximation of this curve can be efficiently computed using standard statistical techniques. This curve is used to determine the amount of slack an activity should have given the basic intuition that the higher the probability of a breakdown occurring before the execution an activity, the greater the amount of slack.

The slack for an activity is computed as a function of the probability that a breakdown will occur before or during the execution of the activity and of the expected breakdown duration. If the $\mu_{tbf}(R)$ for a machine is much less than the scheduling horizon, the possibility of multiple breakdowns must be considered. We do this by considering the cases for each breakdown, nb, separately. First, we assume that at time 0 the machine has just been maintained. For each value

of $nb = 1..M$, where M is a large number, we compute the expected time that the nb-th breakdown will occur as:

$$\mu(nb) = (nb \times \mu_{tbf}(R)) + ((nb-1) \times \mu_{dt}(R)) \quad (4)$$

We calculate the standard deviation of the time for nb breakdowns as:

$$\sigma(nb) = ((nb \times \sigma^2_{tbf}(R)) + ((nb-1) \times \sigma^2_{dt}(R)))^{\frac{1}{2}} \quad (5)$$

These calculations constitute an abuse of the central limit theorem: the random variables representing the breakdown events are not independent. This calculation is an approximation and future work will examine a more sophisticated statistical analysis.

We use the statistics for nb breakdowns to calculate the $P(N(\mu(nb), \sigma(nb)) \leq t)$ curve estimating the probability that nb breakdowns will occur before a particular time t. The amount of slack time required for an activity executing at a particular time point t on resource R is:

$$slack_A(t, R) \geq \sum_{nb=1}^{M} P(N(\mu(nb), \sigma(nb)) \leq t) \times \mu_{dt}(R) \quad (6)$$

As with TWS, we add relation 6 to the problem model as a pruning rule.

6 Experimental Evaluation

To evaluate the three robustness techniques we run a simulation-based experiment. Each problem is solved to optimality: an ordering of the activities on each resource is found that minimizes the sum of the tardiness of the jobs. A simulation of the "execution" of each schedule under uncertainty is then performed.

The problem sets used in our experiments were constructed as follows. Ten 6×6 job shop problems with uncorrelated durations were constructed using a job shop problem generator [WBHW99]. For each problem, TLB, the lower bound on the makespan due to Taillard [Tai93], was calculated. The release dates for each job were assigned by randomly choosing a time (with uniform probability) from the interval $[0, \frac{TLB}{8}]$. Standard temporal propagation was then performed to provide a lower-bound, $dd_{lb}(j)$, on the due date of each job. For each original problem, six problems were then generated by setting the actual due date of each job to $dd(j) = dd_{lb}(j) \times L$, where L represents the "looseness" of the due dates and ranges from 1.0 to 1.5 in steps of 0.1.

For each of these 60 problems, we then introduced nine levels of uncertainty based on two *uncertainty factors*: $U_{machine}$, the number of machines prone to failure and U_{stat}, the magnitude of the uncertainty statistics. Each of the uncertainty factors have three levels {1, 2, 3} and the overall level of uncertainty is, $U = 3 \times (U_{machine} - 1) + U_{stat}$. This encoding produces nine levels of uncertainty divided into three groups. Levels 1-3 have one breakable machine, levels 4-6 have two breakable machines, and levels 7-9 have 3 breakable machines. Within each

group the likelihood of breakdown on each breakable machine increases as the level increases.

The statistical values for each level of U_{stat} are derived as follows for a breakable resource, R. Given the set of activities, $acts_R$, requiring resource R, a lower-bound, $lb(R)$ on the latest end time of the activities is calculated:

$$lb(R) = \max(\min_{A \in acts_R}(est_A) + \sum_{A \in acts_R} dur_A, \max_{A \in acts_R}(lft_A)) \tag{7}$$

Using $lb(R)$, we define, $\mu_{tbf}(R, U_{stat})$, the mean time between failure for resource R, and $\sigma_{tbf}(R)$, the standard deviation time between failure, as follows:

$$\mu_{tbf}(R, U_{stat}) = \frac{lb(R)}{2^{U_{stat}-1}},\ \sigma_{tbf}(R) = \frac{lb(R)}{8} \tag{8}$$

The standard deviation of the down time $\sigma_{dt}(R)$ is simply the mean duration of the activities in $acts_R$ while the mean down time, $\mu_{dt}(R)$ is twice that value.

As we began with 10 problems, 6 values for L, the due date looseness factor, and have a total of 9 combinations of the uncertainty factors, we have a total of 540 test problems.

6.1 Evaluation Criteria

The evaluation of the schedules under uncertainty is done using a simulator. Our optimization function, therefore, has two forms: simulated and predicted. Given problem instance, p, we use $TARD(p, *)$ to denote the minimal sum of the tardiness over all jobs in a predictive schedule. Similarly, we use $TARD(p, s)$ to denote the tardiness of problem instance p in simulation s.

Given a set of simulations, S, and a set of problems, P, the primary basis of comparison of our robustness techniques is the mean simulated tardiness:

$$MST(P, S) = \frac{\sum_{s \in S, q \in Q} TARD(p, s)}{|S| \times |Q|} \tag{9}$$

Our secondary evaluation criteria is the mean absolute difference between the predicted tardiness and the simulated tardiness.

$$MATD(P, S) = \frac{\sum_{s \in S, q \in Q} |TARD(p, s) - TARD(p, *)|}{|S| \times |Q|} \tag{10}$$

6.2 Results

For each test problem and for each robustness technique (including the *no protection* where the uncertainty statistics were ignored), each scheduling problem was solved to optimality using ILOG OPL Studio 3.1, ILOG Scheduler 4.4, and ILOG Solver 4.4. Solving a single problem to optimality took approximately 10 seconds, regardless of robustness technique, on a Pentium II, 300 MHz PC.

A simulator, written in ILOG OPL Studio 3.1, simulated the execution of each schedule, introducing breakdowns based on the specified uncertainty distributions. When a breakdown occurred, the duration of the executing activity was extended by the duration of the breakdown. In the temporal protection condition, the protected durations were replaced with the original durations and the activities were left-shifted (subject to their release dates) before the simulation.

Figure 3 presents the graph of, $MST(P, S)$, the mean simulated tardiness for 100 simulations of each problem under each robustness technique and each combination of uncertainty factors. Except for the highest level of uncertainty, temporal protection results in a higher mean tardiness than is observed even if the uncertainty information is ignored. This is consistent with previous experiments with temporal protection [Gao95]. In contrast, both TWS and FTWS achieve a lower mean tardiness than no protection across all uncertainty levels with FTWS achieving slightly lower mean tardiness than TWS.

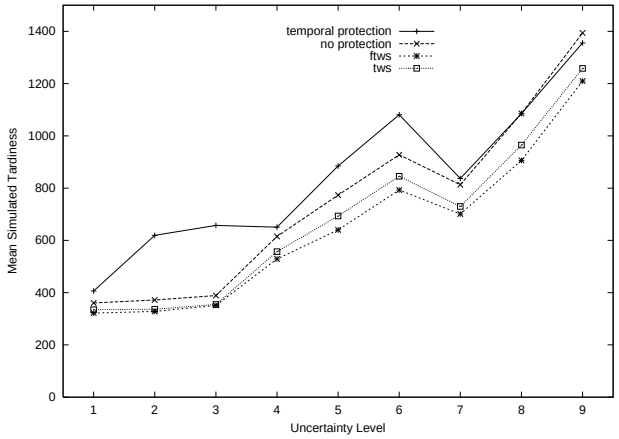


Fig. 3. The mean simulated tardiness for each uncertainty level

Figure 4 presents, $MATD(P, S)$ the mean difference in the absolute value between the simulated tardiness and the predicted tardiness for each robustness technique. Here we observe that for low levels of uncertainty, the predictions of the TWS, FTWS, and no protection techniques are quite similar. In contrast, the temporal protection results vary widely: the mean absolute difference is four times greater than that of the other techniques at uncertainty level 3. As the level of uncertainty increases however, we see the mean absolute difference for no protection increasing quickly while TWS and FTWS results increase more slowly. Interestingly, the relative results of temporal protection improve significantly with increased uncertainty, achieving the lowest mean absolute difference of all techniques at uncertainty levels 7 through 9.

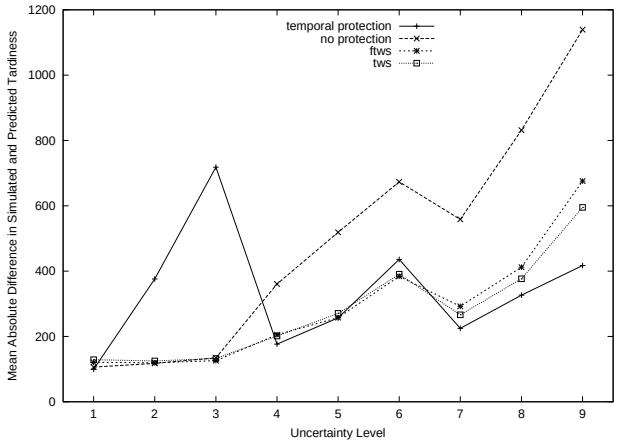


Fig. 4. The mean absolute difference between simulated and predicted tardiness for each uncertainty level

7 Discussion

There are two goals for robustness techniques in scheduling. The first is that though in building robust schedules, the overall schedule quality may be diminished, the accuracy of the predictive schedules is increased. The ability to better predict the actual completion time of a job, even if this completion time is tardy is valuable in real world scheduling. The second goal is that by taking uncertainty into account, the predictive schedule will not only provide more accurate performance information but will actually result in better overall schedule performance. This better performance comes from the fact that the predictive schedule can actually be constructed using the uncertainty information.

In comparing the robustness techniques with ignoring uncertainty information, we see that all techniques achieve the first goal with the exception of temporal protection at low levels of uncertainty. At uncertainly levels above level 3, the absolute difference between the simulated and predicated tardiness (Figure 4) is approximately two times smaller when the uncertainty is taken into account. These differences are not apparent at lower levels of uncertainty and, indeed, temporal protection performs very badly at level 3.

TWS and FTWS also achieve the second goal: Figure 3 shows that the simulated tardiness for the TWS and FTWS solutions is less than that for the solutions with no protection. Except for high level of uncertainty, temporal protection does not result in better overall schedules.

Looking more deeply at the experimental results, we see two interesting phenomenon. First, when only one machine is breakable (i.e., levels 1-3) the no protection condition performs almost as well (and in some cases better) than TWS and FTWS on both mean simulated tardiness and mean absolute difference measures. This is not terribly surprising as, at low levels of tardiness, breakdowns

are less disruptive: unless the breakdown occurs during an activity that is on a critical path[3] some of the breakdown will be absorbed by the naturally occurring slack. Furthermore, with low levels of uncertainty, the level of slack required in the TWS and FTWS conditions is small. The activity sequences in the optimal solutions in the no protection condition will therefore be quite similar to those of TWS and FTWS. Similar sequences will lead to similar simulated tardiness results.

The second phenomenon is that while temporal protection performs very poorly in terms of absolute difference with one breakable machine, when three machines are breakable it has a lower mean absolute difference than TWS and FTWS. The poor behavior, especially at level 3, arises from the fact that extending the durations of the activities on the breakable resource leads to a scheduling problem where the breakable resource is essentially a bottleneck. The optimality of a solution depends almost wholly on the sequence of activities on that resource while the sequences on the rest of the resources are irrelevant. An optimal solution, therefore, has an almost random sequencing of activities on the non-breakable resources. When the duration extensions are removed in the simulation, the sub-optimal sequences on the non-breakable resources leads to high tardiness. In contrast, with multiple breakable machines, optimality depends on more than one resource, leading to a better sequence of activities over more of the resources. The TWS and FTWS methods do not lead to a single bottleneck resource when there is only one breakable machine. This is because the slack that is added to the activities on the breakable resource affects upstream and downstream activities as well. Since by construction all jobs have one activity on the breakable resource, all activities in the problem are constrained to have some level of slack. Even though there is only one breakable resource, all resources are required to have an equal amount of slack and therefore there is no bottleneck resource that completely defines optimality. During problem solving, the activity sequences on the non-breakable resources are just as important as those on the breakable resource in terms of optimality. The fact that the slack is "propagated" to activities that are not on a breakable resource is, in retrospect, obvious. Based on our results, however, it may have a significant impact on the performance of robustness techniques. An interesting question arises as to the relative contribution of reasoning about slack during problem solving and of "slack propagation" toward dealing with uncertainty.

We do not as yet have an explanation of the good performance of temporal protection with high levels of uncertainty.

7.1 Relation to Previous Work

Slack-based techniques involve the addition of extra time in order to recover from unexpected events. Similar approaches, called temporal redundancy, are common in real-time fault tolerant scheduling [GMM95]. Such scheduling problems differ from those typically investigated in the AI community both in the scope (i.e.,

[3] See [Kre00] for a definition of critical path on tardiness minimization problems.

often only one machine) and in the definition of a solution (e.g., a guarantee that the system is schedulable). Nonetheless, real-time fault tolerant scheduling research represents an important source of ideas for further investigations.

Overall, there has been little work in the research literature that specifically addresses uncertainty in the context of the types of scheduling problems that are typically of interest in AI (e.g., problems with multiple resources and activities). A variety of techniques, including resource redundancy [GMM95], probabilistic reasoning [BPLW97,DC97], and a variety of off-line/on-line approaches [Ber93,GB97,Hil94,MHK+98] have been investigated, usually in the context of simpler scheduling problems. There does not yet appear to be a broader understanding of either the role that uncertainty plays in real scheduling problems or a comparison of different approaches.

8 Conclusion

In this paper, we examined three techniques for taking into account uncertainty in scheduling by adding slack to the scheduling problem. Our experiments demonstrate that an existing technique, temporal protection, results in a reduced overall schedule performance but more accurate schedules than not taking uncertainty information into account. The sole exception is at low levels of uncertainty, when temporal protection produces schedules that are significantly less accurate than no protection.

Two novel techniques, time window slack and focused time window slack, were developed to account for the fact that temporal protection reasons about uncertainty as a preprocessing step, before actual scheduling. Time window slack and focused time window slack both incorporate reasoning about uncertainty into the problem solving as well as resulting in a propagation of slack time from activities on breakable resources to temporally connected activities. Our experiments indicate that both the novel techniques are able to produce better, more accurate schedules than either temporal protection or no protection.

We view the work reported in this paper as preliminary. As noted above, there are a number of approaches to uncertainty that have been tried in various types of scheduling problems, however there is not, as yet, any broader understanding of uncertainty as it applies to scheduling problems typically investigated in the AI literature. This paper demonstrates that for a simple, but interesting, class of scheduling problems, slack-based techniques can provide higher quality, more accurate schedules.

9 Acknowledgments

Portions of this research were funded by the Materials and Manufacturing Council of Ontario. Part of this research was performed while the first author was a visiting researcher at SINTEF Applied Mathematics. We would like to thank Geir Hasle, Dag Kjenstad and Martin Stolevik for useful discussions.

References

[Ber93] P. M. Berry. Uncertainty in scheduling: Probability, problem reduction, abstractions, and the user. In *IEE Colloquium on Adcanced Software Technologies for Scheduling*, 1993. Digest No: 193/163.

[BF00] J. C. Beck and M. S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(1):31–81, 2000.

[BPLW97] A Burns, S. Punnekkat, B. Littlewood, and D.W. Wright. Probabilistic guarantees for fault-tolerant real-time systems. Technical Report DeVa TR No. 44, Design for Validation, Esprit Long Term Research Project No. 20072, 1997. Available at http://www.fcul.research.ec.org/deva.

[DC97] R.L. Daniels and J.E. Carrillo. β-robust scheduling for single-machine systems with uncertain processing times. *IIE Transactions*, 29:977–985, 1997.

[Gao95] H. Gao. Building robust schedules using temporal protection–an empirical study of constraint based scheduling under machine failure uncertainty. Master's thesis, Department of Industrial Engineering, University of Toronto, 1995.

[GB97] R.P. Goldman and M.S. Boddy. A constraint-based scheduler for batch manufacturing. *IEEE Expert*, 12(1):49–56, 1997.

[GMM95] S. Ghosh, R. Melhem, and D. Mossé. Enhancing real-time schedules to tolerate transient faults. In *Real-Time Systems Symposium*, 1995.

[Hil94] D. W. Hildum. *Flexibility in a knowledge-based system for solving dynamic resource-constrained scheduling problems.* PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, MA. 01003-4610, 1994. UMass CMPSCI TR 94-77.

[Kre00] S. Kreipl. A large step random walk for minimizing total weighted tardiness in a job shop. *Journal of Scheduling*, 3(3), 2000.

[LP91] C. Le Pape. Constraint propagation in planning and scheduling. Technical report, CIFE Technical Report, Robotics Laboratory, Department of Computer Science, Stanford University, 1991.

[MHK+98] N. Meuleau, M. Hauskrecht, K.E. Kim, L. Peshkin, L.P. Kaelbling, T. Dean, and C. Boutilier. Solving very large weakly coupled markov decision processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.

[MSB88] K.N. McKay, F.R. Safayeni, and J.A. Buzacott. Job-shop scheduling theory: What is relevant? *Interfaces*, 18(4):84–90, 1988.

[SC93] S. F. Smith and C. C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 139–144, 1993.

[Tai93] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

[WBHW99] J.P. Watson, L. Barbulescu, A.E. Howe, and L.D. Whitley. Algorithms performance and problem structure for flow-shop scheduling. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 688–695, 1999.

Dynamic Schedule Management: Lessons from the Air Campaign Planning Domain *

Brian Drabble and Najam-ul Haq

Computational Intelligence Research Laboratory,
1269, University of Oregon,
Eugene, OR 97403
drabble,haqn@cirl.uoregon.edu

Abstract. This paper describes the Dynamic Execution Order Scheduling (DEOS) system that has been developed to handle highly dynamic and interactive scheduling domains. Unlike typical scheduling problems which have a static task list, DEOS is able to handle dynamic task lists in which tasks are added, deleted and modified "on the fly" DEOS is also able to handle tasks with uncertain and/or probabilistic outcomes. DEOS extends the current scheduling paradigm to allow tasking in dynamic and uncertain environments by viewing the planning and scheduling tasks as being integrated and evolving entities. DEOS has been successfully applied to the domains of Air Campaign Planning (ACP) and Intelligence, Surveillance and Reconnaissance (ISR) management. The paper provides an overview of the dynamic task model and the "penalty box" scheduling algorithm which was developed to provide robust solutions to over constrained scheduling problems. The basic algorithm is described together with extensions to handle flexible time constraints.

1 Introduction

This paper describes the Dynamic Execution Order Scheduling DEOS system that has been developed to handle highly dynamic and interactive scheduling problems. Unlike typical scheduling problems which have a static task list, DEOS is able to handle dynamic task lists in which tasks are added, deleted and modified "on the fly". In addition, the dynamic tasking model used by the DEOS system allows it to handle tasks with uncertain and probabilistic outcomes. This allows DEOS to tackle a wider range of problems than possible with previous approaches.

* This research is supported by DARPA Contract: DABT63-98-C-0069 "Intelligent Workflow for Collection Management" and Contract: F30602-97-1-0294 "Understanding and Exploiting Hierarchy". The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of DARPA, Rome Laboratory or the U.S. Government.

Several systems [11, 6, 10] have attempted to solve problems in domains in which there are defined time bounds on activities or where an activity's outcome follows some predictable distribution. For example, in semi-conductor manufacturing a machine may have a failure rate of between 0.1% and 0.5% depending on the chip being manufactured. It may also be the case that some steps need to be re-executed to deal with failures and reworking, e.g. most of the failed chips can be fixed if they pass through steps 112 through 118 again. Systems such as CASPER [3] and CPEF [9] have also attempted to address the planning and execution problem. However, neither system has taken a resource centered optimization approach and neither has attempted to coordinate planning functions across distributed platforms. While these techniques have been successful in domains with limited amounts of uncertainty they are totally unsuitable for dealing with domains such as ACP that contain large amounts of uncertainty (e.g., partially order activities, activities with unknown durations, unexpected outcomes, new requirements) and probabilities (e.g., expected aircraft attrition rates, target damage, locations of enemy forces). The problem is further complicated by the distributed nature of the planning process in which different aspects of the plan are generated and maintained by separate planning cells (e.g. logistics, airborne tankers,, maintenance). The problem becomes one of optimally putting together many different scheduling pieces and monitoring their dependencies and requirements over time.

One of the key aspects of the ACP process is the scheduling of aircraft and weapons to targets (i.e., how many aircraft, of what type, carrying which weapons are assigned to the target). This is a very complex problem as it contains large numbers of different types of constraints (e.g., time, user priority, weight of effort [1], phasing [2], resources). The assignment problems needs to address three major concerns:

1. Identifying trade-offs between different aircraft assignments. For example, a mission's success can be increased if it has fighter escort but these same fighter aircraft could be employed on other bombing missions. If the optimization criteria is to minimize the schedule's makespan and to maximize mission success then the choice of whether or not fighter aircraft are assigned becomes an important trade-off.
2. Identifying the optimal set of targets which can be attacked with the resources available. This requires the scheduler to identify a subset of the targets that can be successfully assigned and to ensure the reasons why targets that are unassigned are fed back to the human planners. This allows for the development of more robust schedules (i.e., ones with a higher probability of succeeding) than previously available to USAF planners. In many cases the human planners would sooner have a schedule that has a high probability of destroying 90% of the targets than one than one which has a low probability

[1] the weight of effort specifies the percentage of aircraft which can be assigned to a particular target type, e.g. 40% of F-15s to SAM targets.

[2] phasing specifies the relative order target types should be attacked, e.g. all SAMs before bridges

of success but attacks 100% of the targets. The problem is identifying what percentage is possible and the targets in that sub-set. In addition, it is vital to avoid situations where many missions must be canceled or replanned because of small anomalies, such as a single target being missed.

3. Identifying the optimal break point at which the air campaign should switch from one target type to another. By finding the optimal break point it becomes possible to assign resources to attack high priority targets in temporally later target sets rather than using limited resources trying to destroy all the targets in an earlier target set the last few of which have relatively low value. Again this allows for more robust schedules which have a higher probability that they will achieve their overall aims.

The key to DEOSs ability to successfully solve problems in this domain is that it can generate schedules very quickly and be adaptable to changes in the task and the situation. There is no point in DEOS generating schedules for the next 12 hours when the schedule needs to change on a minute by minute basis. The core algorithm of the DEOS system is the "Squeakywheel" optimization (SWO) technique developed by Joslin and Clements [7]. The basic SWO algorithm has been modified to handle several new constraint types and these include probability distributions, probabilistic functions, temporal windows, resource limits and a limited set of precedence constraints. In addition a more expressive task description language [2, 1] has been integrated to allow the scheduler to better model the actual dynamics and activities in the domain. The algorithm has also been modified to allow it to identify optimal sub-sets of tasks from the task list and this technique is referred to as Penalty Box scheduling. These modification are generic and could be easily applied to problems in manufacturing, assembly, integration and test. Details of the task model and algorithm modifications are provided later in the paper.

Previous work [8] has addressed aspects of this problem but this approach differs in several important ways. The overall DEOS approach is to identify optimal resource assignments and where insufficient resources are available the best sub-set. The previous work [8] took an MDP approach to try and identify the best policy for a given target. This resulted in a solution in which the target may need to be attacked for several days consecutively and discussions for USAF pilots have shown that such a mission plan is usually a suicide one.[3] The DEOS approach is able to handle problems far larger and generate solutions in a few seconds as opposed to tens of minutes. In addition, the DEOS approach is able to handle a richer set of constraint types and optimization criteria (e.g., minimize makespan, maximize probability of damage and minimize attrition). Finally, the DEOS approach is able to handle the dynamic aspects of the problem (e.g. missed targets, pop-up targets) which the previous work cannot. This allows DEOS to develop schedules which are robust against certain types of change and minimize the knock on effects of changing missions on the fly. Current USAF planning systems use LP/IP solvers to generate mission schedules. The core SWO algorithm

[3] The enemy begin to expect the raids and hence the attrition rate becomes very high!!

has been compared with LP/IP solvers on several manufacturing problems and was found to out perform them in terms of the speed of solution and the quality of the solutions generated [4].

The paper is structured as follows, Firstly, it provides an overview of the ACP domain and the data used by the DEOS system. Secondly, it provides an overview of the task model and thirdly, it describes the basic scheduling algorithm and two extensions which allow it to identify optimal sub-sets. Fourthly, it provides details of the schedules generated and their evaluation by members of the USAF. Finally, it provides a summary of current progress and describes several additional techniques and ideas which will be explored.

1.1 Overview of the Target Scheduling Process

The weapon/aircraft pairing problem is a complicated one due to the many different trade-offs which are possible and the ability of the aircraft to be configured to suit different missions and different weapons loads. The actual weapon/aircraft pairing is based on a set of probabilities which take into account, probability of hitting the target, destroying the target[4] and the expected attrition rate of the aircraft against the target type. In theory any weapon/aircraft pairing could be sent against a target but it may have a very low chance of success. As described earlier some aircraft have a greater probability of success if additional assets are sent with them. For example, the expected aircraft attrition rate can be reduced by sending SEAD aircraft with the strike aircraft. However, this would mean that the SEAD aircraft could not be used as strike aircraft which may result in their being insufficient resources to attack a high value target later in the schedule. In addition to the constraints on individual targets and aircraft there are also further constraints relating to time and resource limits. The temporal constraints specify a window during which a target must be attacked, the window during which targets of a particular type can be attacked and the time delay between targets which are "connected" (e.g. the cooling towers of a power station must be attacked with 12 hours of the generator halls). The resource constraints specify the available quantities of aircraft and weapons (which can vary over time) and percentages limits on the number of aircraft which can assigned to a given target type[5] (e.g. 40% of missions against air defenses, 20% of missions against communication sites). These constraints are very problematic as the number of missions is not known in advance hence the scheduler needs to keep the percentages of different mission types in balance. The targets themselves are grouped into target sets (e.g. all bridges across the Thames) and these are then grouped into target systems (e.g. all railway centers in southern England). Unfortunately, the same target might be in two or more different target sets and hence has a higher "value" than the other targets in the same set. In addition, it may be the case that it is not necessary to attack all the targets in the set to achieve

[4] some weapons may be able to hit a target but not destroy it, e.g. an anti-tank missile can hit a building but it very unlikely to destroy it

[5] This is referred to as the Weight of Effort.

the overall aim. For example, if the aim is to stop the enemy forces crossing the river it may be possible to achieve this by destroying only 80% of the bridges. This makes target selection a very important aspect of the scheduling process. The scheduling process aims to find an optimal assignment of aircraft to targets which minimizes the probability of needing to restrike the target or cause collateral damage while also minimizing the risk to the assigned aircraft.

The problem is further complicated by the fact that aircraft can be reassigned to a different mission on the fly. For example, aircraft could be diverted to attack a pop-up target for which they are an optimal match. Alternatively, the aircraft may be a good match but the weapons they are carrying are not. This means the aircraft could be diverted to a base and reconfigured if time permits. Changing missions on the fly has potential knock on effects with later missions being postponed or reassigned due to longer than expected mission durations.

1.2 Mission Planning Data Models

The target matching problem is driven by a set of tables which provide details of the different aircraft, weapons, targets, support assets, etc. The primary information source is the target table and a section is provided in Table 1. This table show the type of mission, air superiority (AS) the hardness of the target and the risk associated with attacking the target [6]. Associated with each target type is a reward which defines the importance of the target to the human planners. Table 2 shows a section of the rewards table (these values were calculated through discussions with human planners and through the analysis of the schedules generated by DEOS).

Obj	Task	Lat	Long	Hardness	Risk	Target
AS	1	180804N	233902W	Hard	High	airfield
AS	3	172708N	223930W	Soft	Medium	radar-comms
...						

Table 1. ACP Target Table

The mission type from Table 1 identifies the class of aircraft which could be sent against the target. Table 3 shows the mapping of aircraft to mission type and shows that the same aircraft can be used in for many different missions.

One of the optimization criteria for this problem is minimize risk and DEOS tries to identify aircraft which have a low risk against a selected type of target. The expected risk to an aircraft is calculated by summing the total probability that the aircraft will be shot down either to or from the target[7] One key decision

[6] Not shown is the time window during which the target needs to be attacked, e.g. D+5, D+10, etc

[7] SAM batteries have a threat radius which has a known probability of detection based on the distance the aircraft is from the center of the radius.

Mission	Reward
mil activity	45
SAM site	90
SSM site	100
C3	35
command HQ	60
...	

Table 2. Target Reward Table

DEOS needs to make is to whether or not to use SEAD protection to reduce the risk to the attacking aircraft. As pointed out earlier this may have the side effect of making another strike mission late.

Once a potential aircraft has been selected it must be checked to ensure that it can carry appropriate weapon load for the target. The probability of destruction is noted in terms of a single weapon and the current USAF doctrine is that the plane carries enough weapons to give a 90% or better chance of destroying the target. There is no guarantee that a specified weapon load will destroy the target as they could all miss. Additional tables provides details of the probability of weapon hitting the target, provide data on air to air refueling times, aircraft speed and range, turn round times, etc. Full details of the aircraft/weapon pairing algorithm are given later in the paper.

Mission	Aircraft
Counter Air	F15E, F117, F16C, F16CLN, F14A, F14B, F14D, FA18A, FA18C, FA18D, AV8B, B52H, B1B
SEAD	F16CJ, EA6B, FA18C
Def Counter Air	F14A, F14B, F14D, F16C
Interdiction	F15E, F117, F16C, F16CLN, B52H, B1B, F14A, F14B, F14D, FA18A, FA18C, FA18D, AV8B
Close Air Support	A10, AV8B, F16C, F15E, FA18A, FA18C, FA18D
Strategic Attack	F117, B52H, B1B

Table 3. Aircraft and Mission Mapping Table

Each mission is modeled using the PRFER mission task model [5] that defines a natural breakdown of a mission into its constituent parts or sub-blocks.

- **P**lan: Time taken for the pilot to plan the mission. Once a plan has been identified it is inserted in the slot for other workflow tasks to examine and check.
- **R**eady: Time taken to prepare the plane for the mission

– **F**ly: Time taken to get to the mission objective [8]
– **E**xecute: Time taken to execute the mission, e.g. drop weapons, unload food pallets, etc.
– **R**econstitute: Time taken to turn the aircraft round once it has returned to base.

The PRFER model allows a tasking agent to create a better model of the processing the task needs and to better understand how to allocate resources, identify tradeoffs, asses changes and modify the associated task list. The sub-blocks are allowed to "breath" as changes in the domain are reflected as changes in one or more of the sub-blocks. For example, if the aircraft chosen for the mission develops a failure during its ready time then the "Ready" sub-task will expand and accommodate the extra time. The PRFER model allows DEOS to quickly identify the impact of changes, propose potential changes to the mission tasking and inform the planners of new deadlines and constraints (e.g. the planes now on hold need refueling in the next 30 minutes).

2 Resource Allocation Algorithm

The basic concept behind DEOS is to generate schedules quickly and to update them on the fly as new requirements and changes occur in the domain. The core SWO algorithm uses a priority queue to determine the order in which tasks should be released to a greedy scheduling algorithm. This identifies the best aircraft/weapon for a given task from those available. Tasks later in the priority queue have a smaller choice of resources due to earlier commitments. The order of the priority queue is determined by how difficult the task is to deal with that is, the higher the task is in the queue the harder it is to handle it correctly. It does not require an external priority to be identified by the user. Once a schedule has been generated it is analyzed to identify which tasks were handled badly (e.g., a task was completed after its deadline, or assigned to a high attrition aircraft). Any task that "squeaks" (i.e., was handled badly) is given a "blame score" and is promoted in the priority queue, with the distance it is promoted determined by the extent of the problem. This new priority queue is then used to generate another schedule that is analyzed for problems. This process continues until no significant improvement in the schedule is noted over several iterations. SWO is extremely fast with each cycle of generate, analyze, and re-prioritize taking only a few seconds, even for large problems.

One of the key issues in this domain was to generate schedules which balanced a number of potentially conflicting factors. For example, the planners wanted all 2500 targets attacked in the shortest time, with minimum attrition and minimum risk of collateral damage. However, to guarantee that each target was attacked with minimum risk would require all missions to be flown by F-117s and that would result in very long schedules. A sample schedule was generated which used

[8] This can be replaced by a "drive" or "sail" block for operations using land or sea transport

only the best target/aircraft pairing and it had a makespan in excess of six days. Using the DEOS approach the schedule was reduced in length to just under two days with a less than 1% reduction in overall schedule quality.

To address these potential conflicts a series of functions were developed which investigated the different aspects of the problem, e.g. aircraft attrition, probabilities of hitting and destroying the target, numbers of weapons needed, number of aircraft needed, support assets, number of sorties, etc. It was identified that the key elements of evaluation were the probability that the target would be attacked successfully and that the attacking aircraft would have a low attrition rate. This allowed two main functions to be identified [9]. Function 1 describes the probability that a target will be destroyed given a specified number of weapons `W` and the probabilities of hit and kill (P_h P_k) respectively for a single weapon. Function 2 describes the expected attrition rate for `n` aircraft when attacking with `N` total aircraft.

Function 1:

$$P_{kill}(W, P_h, P_k) =$$
$$P_h^W \sum_{n=1}^{w} \binom{w}{h} \left(\frac{1-P_h}{P_h}\right)^{w-n} \left(1-(1-P_k)^N\right)$$

Function 2:

$$P_{attrition}(N, n, P_a) = \binom{N}{n} (1-P_A)^{N-n} P_A^n$$

The DEOS uses these formulas to evaluate different combinations of weapons and aircraft for a given target type, trying to identify the best possible match. However, it may be the case that the required aircraft/weapon pairing may be unavailable in the desired time interval (e.g., between 0900 hrs and 1100 hrs all F-16s may be assigned to other missions). DEOS may decide to use a second option (i.e., a different aircraft and/or weapon) and will cycle through the different options until an assignment of aircraft/weapons to the target can be made [10]. In addition, DEOS may add in a SEAD sortie to off set a high expected attrition rate. After an assignment has been made it may be the case that it is a poor one (e.g., high attrition rate, low probability of success) and this is dealt with in the next cycle of algorithm when the generated schedule is analyzed and poor assignments identified.

During the development of the algorithm it was identified that in many cases the number of targets greatly exceeded the available resources. In addition, it was also identified that some of the time constraints provided by the human schedulers were leading to less than optimal schedules. Details of the modifications to the basic algorithm are provided in the following sections.

[9] Other support functions were developed but are not discussed in this paper

[10] By default the targeting database provided 5 options but the 4th and 5th usually had a low probability of success

2.1 Penalty Box Scheduling

The aim of penalty box scheduling is to identify a sub-set of tasks which can be resourced effectively and avoid the problem of generating low quality schedules which resource all tasks. For example, human planners may be happier striking 90% of the targets with high probability of success rather than 100% of the targets with a much lower probability of success (i.e., the planners wanted robust solutions which has a higher probability of success). The problem is finding what percentage can be assigned and which tasks to select. Penalty box scheduling extends the SWO algorithm by viewing the inability to assign a task within its specified time window as a high priority problem (i.e., a large squeak). Instead of placing the task at a point later in the schedule the task is put in the penalty box[11] for a single cycle of the algorithm. The penalty tasks are assigned a high blame value and their position in the priority queue altered. The blame value also takes into account the potential reward for striking the target and the external priority assigned by the user to the target set. At the end of the scheduling process[12] those tasks in the penalty box are left unassigned. This extension proved highly efficient (i.e., there was a negligible slow down in the speed of solution) at identifying sub-sets of tasks and provided the human schedulers with more robust solutions to the targeting problem. After scheduling was complete the human planners were able to provide feedback on which tasks left in the penalty box needed to be resourced. They could then compare the resulting schedule with the optimal one and measure (i.e., number of missions, sorties, expected attrition rate, etc) the drop in the overall schedule quality.

2.2 Temporal Phase Transition

Missions are specified with time windows during which the mission must be accomplished. However, these associated time windows tend to be arbitrary and estimates by the human planners. Rather than use the time constraints as invariable, DEOS was allowed to relax them and attempt to identify the point at which to switch from one mission type to another. For example, attacking SAM sites should be completed first (for the next 6 hours) and then attacks against power stations for the next 6 hours. Their division may mean that fairly low priority SAM missions can be handled whereas only the highest priority power station missions can be assigned. A better schedule may be to stop SAM missions after 4 hours and give the additional 2 hours to the power station missions. The selection of suitable subsets needs to be weighted against the flexibility built into the schedule by allocating maximal windows. For example, more tasks might be resourced within a window at the expense of making the schedule more brittle.

The temporal phase transition problem was investigated through two different methods. The first method involves a variation of the penalty box scheduling

[11] This is a term connected with sports where a player committing an offense is placed in the penalty box for a specified period.

[12] DEOS keeps track of the best schedule found so far and its associated penalty box entries.

algorithm in which pointers are maintained to the last task of the temporally earlier set and to the earliest task of the later set respectively. It always the case that no SAM can be placed after any power mission. For example, if a SAM mission cannot be scheduled before the earliest power mission then it is sent to the penalty box for a cycle. Alternatively, if a power mission can be scheduled after all SAM missions but before the current earliest power mission then it can be added and the pointers updated. This relies on the ability of the critiquing phase of the SWO algorithm to apportion blame appropriately to move the missions in the penalty box the required distance in the priority queue. The second method involves rippling all the power missions to the right to fit in a new SAM mission. Any power missions already in the schedule keeps their assignment (i.e. a F-16) but are moved later in time (i.e., they do not have to accept a lower quality assignment). If the tasks cannot be rippled right then the new task is assigned to the penalty box. This relies on the construction phase of SWO algorithm being able to reconstruct new partial assignments on the fly. By having already assigned power tasks keep their assignments (or be assigned one no worse (i.e., swap the F-16 or a F-15) it keeps the problem tractable. The analysis of the schedule showed that on problem sizes up to 2000 tasks it was better to use the shuffle approach and for problems greater than 2000 the pointer approach was marginally better.

3 Results

Figure 1 shows the performance of DEOS on an example test set of 700 targets and 150 aircraft. The optimization criteria included low attrition rate, high probability of success and a minimal makespan. The best schedule identified completes all 700 targets in 47 hours with an expected loss rate of less than 1%. To date the DEOS results are the best for these problem and easily surpass those developed by current USAF mission planners. Figure 1 shows that the addition of penalty box scheduling and phase transition components does not effect the overall performance of the system. DEOS very quickly settles in an appropriate region of the search space and spends many iterations trying to improve on a reasonably good schedule. DEOS is trying to identify trade-offs between the different optimization criteria and Table 4 shows a typical example. Between iteration 2 and 3 the `raw` score[13] increased by less than 1% but the `analysis` score[14] increased by nearly 25% due to the schedule being a lot shorter.

The example above also shows that DEOS was able to find the best sub-set of targets from those specified (e.g., 667 out of 700 were successfully tasked). The DEOS schedules allow USAF planners to identify robust solutions and the incremental costs (e.g., additional planes, sorties, attrition) necessary to attack all targets. For each target DEOS identifies an appropriate number of aircraft, weapon load and timing information. In some cases the assigned aircraft/weapon

[13] This is the summation of the number of targets attacked, probability of success, number of missions and sorties

[14] This is the raw score divided by the makespan in minutes

Iteration	Targets Assigned	Raw Score	Analysis
2	667	667075	15744
3	667	669873	20219

Table 4. Target set vs Makespan Trade-off

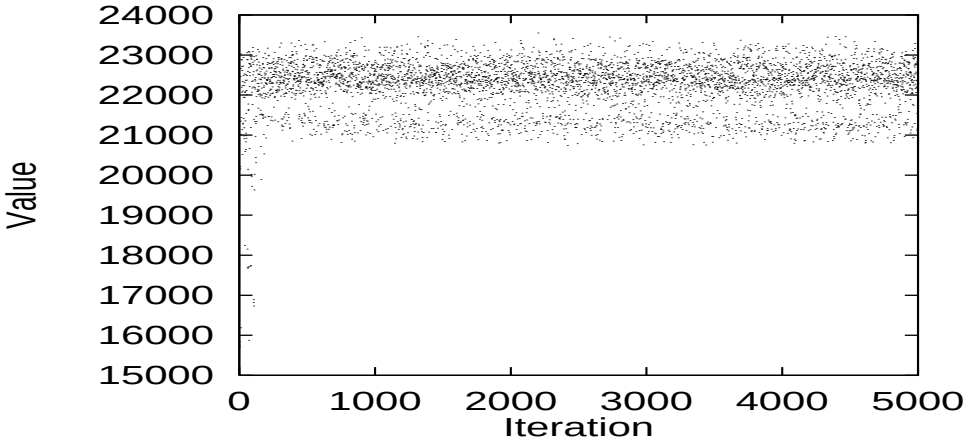


Fig. 1. Air Mission Planning Results

is a less than optimal match. It is often the case that to obtain a good overall schedule some tasks need to be handled badly (i.e., they need to be sacrificed). It is possible to handle the sacrificed tasks better but only at the expense of making the overall schedule worse. The presence of "sacrifice tasks" usually indicates that additional resources of a particular class are needed. The system was evaluated by subject matter experts (SME) from the USAF. The aim was to show that the SME's view of schedule quality and that of DEOS were correlated. The SMEs were given pairs of schedules whose difference in quality narrowed gradually and were asked to choose the better schedule. In all cases the view of the SME and DEOS was correlated. After six iterations the SMEs were unable to make an informed decision over which schedule was better.

4 Summary and Further Work

This paper has presented a description of the DEOS scheduling system, its scheduling algorithm and its application to the mission scheduling problem. DEOS allows for the explicit analysis of trade-offs in resource allocation, dynamic update of on going schedules, on the fly task addition and for focussed impact analysis and repair. To date the system has been applied to large scale ACP problems (i.e. 2500 targets and 200 aircraft over a 5 day period) and was successfully demonstrated as part of the USAFs Effects Based Operations project at the end of 2000. The techniques are generalizable to other domains in which there are

flexible time constraints and the "penalty box" techniques are applicable to problems where there is phasing between different groups of tasks. For example, in manufacturing domains schedulers are often faced with the problem of switching production from one type to another to improve overall productivity. Several improvements will be made to DEOS and these include adversarial planning in which the schedule will propose robust solutions to potential enemy responses. The interface will be improved to allow easier interaction and specification of policies and preferences. The results from the ACP domain and other non-probabilistic manufacturing domains show a distinct grouping of schedule quality as shown in Figure 1. These groups represent classes of solutions (rather than point solutions) that have particular attributes and values. DEOS will be modified to automatically identify these discontinuities in the solution space and alert the planners.

References

1. Berry, P.M. and Drabble, B: SWIM: An AI-based System for Workflow Enabled Reactive Control, in the Proceedings of the Workshop on Workflow and Process Management held as part of the International Joint Conference on Artificial Intelligence (IJCAI-99), (eds, B, Drabble and M. Ibrahim), IJCAI Inc, August, 1999.
2. Berry, P.M. and Drabble, B.,: The AIM Process Modeling Methodology, AI Center, SRI International, Technical Report, 1789, Menlo Park, CA, 1999.
3. Chien, S., Knight, R., Stechert, A., Sherwood, R., and Rabideau, G.,: Integrated Planning and Execution for Autonomous Spacecraft, in the proceedings of the IEE Aerospace Conference (IAC), Aspen, CO, March 1999.
4. Clements, D., Crawford, L., Joslin, D., Nemhauser, G., Puttlitz M. and Savelsbergh, M.: Heuristic Optimization: A hybrid AI/OR approach, in the proceedings of the Workshop on Industrial Constraint-Directed Scheduling, 1997. (Held in conjunction with CP'97, Schloss Hagenberg, Austria.).
5. Drabble, B.: Task Decomposition Support to Reactive Scheduling, in the proceedings of the 5th European Conference on Planning (ECP-99), Springer Verlag Press, New York, NY, USA, September, 1999.
6. Fox, M.S.: ISIS: A Retrospective, in Intelligent Scheduling, (Zweben. M. and Fox, M.S.), 1994. Publisher Morgan Kaufmann, Palo Alto, CA, 94303, USA, pp3-28.
7. Joslin, D.E. and Clements, D.P.: Squeakywheel Optimization, in the proceedings of the Fifteenth National Conference on Artificial Intelligence, Madison, WI, AAAI Press, Menlo Park, CA, USA 1998.
8. Meuleau, N., Hauskrecht, M., Kim K., Peshkin. L., Pack Kaelbling. L., Dean. T., and Boutilier., C.: Solving Very Large Weakly Coupled Markov Decision Processes, in the Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, MIT, Cambridge, MA 02142, USA, July, 1998, pp165-172.
9. Myers, K.L..: A Continuous Planning and Execution Framework, AI Magazine, Vol 20(4), AAAI Press, Menlo Park, CA, 1999.
10. Sadeh, N.: Micro-Opportunistic Scheduling: The Micro-Boss Factory Scheduler, in Intelligent Scheduling, (Zweben. M. and Fox, M.S.), 1994. Publisher Morgan Kaufmann, Palo Alto, California, 94303, USA, pp99-136.
11. Smith, S.F.: OPIS: A Methodology and Architecture for Reactive Scheduling, in Intelligent Scheduling, (Zweben. M. and Fox, M.S.), 1994. Publisher Morgan Kaufmann, Palo Alto, CA, 94303, USA, pp29-66.

Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results

Philippe Laborie
ILOG S.A., 9, rue de Verdun, BP 85
F-94253 Gentilly Cedex, France
plaborie@ilog.fr

Abstract. This paper summarizes the main existing approaches to the propagation of resource constraints in Constraint-Based scheduling and identifies some of their limitations for using them in an integrated planning and scheduling framework. We then describe two new algorithms to propagate resource constraints on discrete resources and reservoirs. Unlike most of the classical work in scheduling, our algorithms focus on the precedence relations between activities rather than on their absolute position in time. They are efficient even when the set of activities is not completely defined and when the time window of activities is large. These features explain why they are particularly suited for integrated planning and scheduling approaches. All our algorithms are illustrated with examples. Some encouraging preliminary results are reported on pure scheduling problems.

1 Introduction

As underlined in [18], some tools are still missing to solve problems that lie between pure AI planning and pure scheduling. Until now, the scheduling community has focused on the optimization of big scheduling problems involving a well-defined set of activities and resource constraints. In contrast, AI planning research - due to the inherent complexity of plan synthesis - has focused on the selection of activities leaving aside the issues of optimization and the handling of time and complex resources. From the point of view of scheduling, mixed planning and scheduling problems have two original characteristics. First, as the set of activities is not completely known beforehand it's better to avoid taking strong scheduling commitments during the search (e.g. instantiating or strongly reducing the time window of an activity). Secondly, most of the partial plans handled by partial order planners (POP) or by hierarchical task network planners (HTN) make an extensive usage of precedence constraints between activities. And, surprisingly, until now the conjunction of precedence and resource constraints has not been deeply investigated, even in the scheduling field itself. Indeed, except for the special case of unary resources (for example in job-shop scheduling), disjunctive formulations of cumulative resource constraints are relatively new techniques and until now, they were mainly used for search control and heuristics [5,14]. This paper proposes some new constraint propagation algorithms that strongly exploit the conjunction of precedence and resource constraints and allow a natural implementation of least-commitment planning and scheduling approaches. The first section of the paper describes our scheduling model. The second one summarizes the state-of-the-art scheduling propagation techniques and explains why most of them are not satisfactory for dealing with integrated

planning and scheduling. In the next section, we describe the basic structure which the new algorithms we propose rely on: precedence graphs. Then, we present two original techniques for propagating resource constraints: the energy precedence algorithm and the balance algorithm. Finally, the last section of the paper describes how these propagation algorithms can be embedded in a least-commitment search procedure and gives some preliminary results on pure scheduling problems.

2 Model and Notations

Partial schedule. A **partial schedule** corresponds to the current scheduling information available at a given node in the search tree. In a mixed planning and scheduling problem, it represents all the temporal and resource information of a partial plan. A partial schedule is composed of activities, and temporal constraints and resource constraints. These concepts are detailed below.

Activities. An **activity** A corresponds to a time interval $[start(A), end(A))$ where $start(A)$ and $end(A)$ are decision variables denoting the start and end time of activity A. $start_{min}(A)$, $start_{max}(A)$, $end_{min}(A)$ and $end_{max}(A)$ will respectively denote the current earliest start time, latest start time, earliest end time and latest end time of activity A. The duration of activity A is a variable $dur(A) = end(A) - start(A)$. Depending on the problem, the duration may be known in advance or may be a decision variable. In a mixed planning and scheduling problem, a planning operator may be composed of one or several activities.

Temporal constraints. A **temporal constraint** is a constraint of the form: $min \leq t_i - t_j \leq max$ where t_i and t_j are either some variable representing the start or end time of an activity or a constant and min and max are two integer constants. Note that simple precedence between activities as well as release dates and due dates are special cases of temporal constraints.

Resources. The most general case of resources we consider in this paper is the reservoir resource. A **reservoir** resource is a multi-capacity resource that can be consumed, produced and/or just required over some time interval by the activities in the schedule. A reservoir has an integer maximal capacity and may have an initial level. As an example of a reservoir, you can think of a fuel tank. A **discrete resource** is a reservoir resource that cannot be produced. Discrete resources are also often called renewable or sharable resources in the scheduling literature. A discrete resource has a known maximal capacity that may change over time. A discrete resource allows for example to represent a pool of workers whose availability varies over time. A **unary resource** is a discrete resource with unit capacity. It imposes that all the activities requiring the same unary resource are totally ordered. This is typically the case of a machine that can only process one job at a time. Unary resources are the simplest and the most studied resources in scheduling as well as in AI planning.

Resource constraints. A **resource constraint** defines how a given activity A will require and affect the availability of a given resource R. It consists of a tuple $< A, R, q, TE >$ where q is an integer decision variable describing the quantity of resource R consumed (if $q < 0$) or produced (if $q > 0$) by activity A and TE is a time extent that describes the time interval where the availability of resource R is affected by the execution of activity A. For example:

- $< A, R_1, -1, FromStartToEnd >$ is a resource constraint that states that activity A will require 1 unit of resource R_1 between its start time and its end time.
- $< A, R_2, q = [2,3], AfterEnd >$ is a resource constraint that states that activity A will produce 2 or 3 units of reservoir R_2 at its end time. This will increase the availability of R_2 after the end time of A.
- $< A, R_3, -4, AfterStart >$ is a resource constraint that states that activity A will consume 4 units of resource R_3 at its start time. This will decrease the availability of R_3 after the start time of A.

Of course, the same activity A may participate into several resource constraints. Note that the change of resource availability at the start or end time of an activity is considered to be instantaneous: we do not handle continuous changes.

Close Status of a Resource. At given node in the search, we say that a resource is **closed** if we know that no additional resource constraint on that resource will be added in the partial schedule when continuing in the search tree. In stratified planning and scheduling approaches where the planning phase is separated from the scheduling one, all the resources can be considered closed during scheduling as all the activities and resource constraints have been generated during the planning phase. Note also that in approaches that interleave planning and scheduling and implement a hierarchical search as in [11], it is also possible to identify as closed resources during the search the ones belonging to already processed abstraction levels.

3 Existing Approaches

From the point of view of Constraint Programming, a partial schedule is a set of decision variables (start, end, duration of activities, required quantities of resource) and a set of constraints between these variables (temporal and resource constraints). A solution schedule is an instantiation of all the decision variables so that all the constraints are satisfied. In Constraint Programming, the main technique used to prune the search space is **constraint propagation**. It consists in removing from the domain of possible values of a decision variable those values that we know for sure will violate some constraint. More generally, constraint propagation allows finding in the current problem some features shared by all the solutions reachable from the current search node; these features may be some domain restriction or some additional constraints that must be satisfied. Currently, in constraint-based scheduling there are two families of algorithms to propagate resource constraints: timetabling approaches and activity interaction techniques.

3.1 Timetabling

The first propagation technique, known as **timetabling**, relies on the computation for every date t of the minimal resource usage at this date by the current activities in the schedule [7]. This aggregated demand profile is maintained during the search and it allows restricting the domains of the start and end times of activities by removing those dates that would necessarily lead to an over-consumption of the resource. For simplicity reason, we describe this technique only on discrete resources and assuming all the time extents are $FromStartToEnd$. Suppose that an activity A requires $q(A) \in [q_{min}(A), q_{max}(A)]$ units of a given resource R and is such that $start_{max}(A) < end_{min}(A)$, then we know for sure that A will at least execute between

$start_{max}(A)$ and $end_{min}(A)$ and thus, it will require for sure $q_{min}(A)$ units of resource R on this time interval. For each resource R, a curve is maintained that aggregates all these demands that is:

$$C_R(t) = \sum_{\{<A,R,q,TE>/start_{max}(A) \leq t < end_{min}(A)\}} q_{min}(A)$$

It's clear that if there exists a date t such that $C_R(t)$ is strictly greater than the maximal capacity of the resource Q, the current schedule cannot lead to a solution and the search must backtrack. Furthermore, if there exists an activity B requiring $q(B)$ units of resource R and a date t_0 such that: $end_{min}(B) \leq t_0 < end_{max}(B)$ and $\forall t \in [t_0, end_{max}(B)), C_R(t) + q_{min}(B) > Q$ then, activity B cannot end after date t_0 as it would over-consume the resource. Indeed, you must remember that, as $end_{min}(B) \leq t_0$, B is never taken into account in the aggregation on the time interval $[t_0, end_{max}(B))$. Thus, t_0 is a new valid upper bound for $end(B)$. A similar reasoning can be applied to find new lower bounds on the start time of activities as well as new upper bounds on the quantity of resource required by activities. Moreover, this approach can easily be extended to all types of time extent and to reservoirs. The main advantage of this technique is its relative simplicity and its low algorithmic complexity. It is the main technique used so far for scheduling discrete resources and reservoirs. Unfortunately, these algorithms propagate nothing until the time windows of activities become so small that some dates t are necessarily covered by some activity. It means that unless some strong commitments are made early in the search on the time windows of activities, these approaches are not able to efficiently propagate. Furthermore, these approaches do not directly exploit the existence of precedence constraints between activities.

3.2 Activity Interactions

The second family of algorithms is based on an analysis of **activity interactions**. Instead of considering what happens at a date t, it considers some subsets Ω of activities competing for the same resource and performs some propagation based on the position of activities in Ω. Some classical activity interaction approaches are summarized below.

Disjunctive Constraint. The simplest example of such an algorithm is the disjunctive constraint on unary resources [8]. This algorithm analyzes each pair of activities (A, B) requiring the same unary resource and, whenever the current time bounds of activities are so that $start_{max}(A) < end_{min}(B)$, it deduces that as activity A necessarily starts before the end of activity B is must be completely executed before B and thus, $end(A) \leq start_{max}(B)$ and $start(B) \geq end_{min}(A)$. Actually, the classical disjunctive constraint can be generalized as follows: whenever the temporal constraints are so that the constraint $start(A) < end(B)$ must hold, it adds the additional constraint that $end(A) \leq start(B)$. Note that this algorithm is the exact counterpart in scheduling of the disjunctive constraint to handle unsafe causal links in POCL planners proposed in [13]. Unfortunately, such a simple constraint only works in the restricted case of unary resources.

Edge-Finding. Edge-finding techniques [3, 16] are available for both unary and discrete resources. On a unary resource, edge-finding techniques detect situations where a given activity A cannot be executed after any activity in a set Ω because there would not be enough time to execute all the activities in $\Omega \cup A$ between the earliest start

time of activities in $\Omega \cup A$ and the latest end time of activities in $\Omega \cup A$. When such a situation is detected, it means that A must be executed before all the activities in Ω and it allows to compute a new valid upper bound for the end time of A. More formally, let Ω be a subset of activities on a unary resource, and $A \notin \Omega$ another activity on the same unary resource. Most of the edge-finding technique can be captured by the rule (1) $\Rightarrow$ (2) where:

$$(1)\ end_{max}(\Omega \cup A) - start_{min}(\Omega) < dur(\Omega \cup A)$$
$$(2)\ \ end(A) \leq \min_{\Omega' \subset \Omega} (end_{max}(\Omega') - dur(\Omega'))$$

Similar rules allow to detect and propagate the fact that a given activity must end after all activities in Ω (*Last*), cannot start before all activities in Ω (*Not First*) or cannot end after all activities in Ω (*Not Last*). Furthermore, edge-finding techniques can be adapted to discrete resources by reasoning on the resource energy required by the activities that is, the product *duration* $\times$ *required quantity*. Most of the edge-finding algorithms can be implemented to propagate on all the activities A and all the subsets Ω with a total complexity in $O(n^2)$.

Energetic Reasoning. As for the edge-finding techniques, energetic reasoning [9] analyzes the current time-bounds of activities in order to adjust them by removing some invalid values. A typical example of energetic reasoning consists in finding pairs of activities A, B on a unary resource such that ordering activity A before B would lead to a dead-end because the unary resource would not provide enough "energy" between the earliest start time of A and the latest end time of B to execute A, B and all the other activities that necessarily needs to execute on this time window. More formally, if C is an activity and $[t_1, t_2)$ a time window, the energy necessarily required by C on the time window $[t_1, t_2)$ is:

$$W_C^{[t_1,t_2)} = min(end_{min}(C) - t_1, t_2 - start_{max}(C), dur(C), t_2 - t_1)$$

Thus, as soon as the condition below holds, it means that A cannot be ordered before B and thus, must be ordered after. It allows to update the earliest start time of A and the latest end time of B.

$$end_{max}(B) - start_{min}(A) < dur(A) + dur(B) + \sum_{C \notin \{A,B\}} W_C^{[start_{min}(A), end_{max}(B))}$$

Other adjustments of time bounds using energetic reasoning exist that allow, for example to deduce that an activity cannot start at its earliest start time or cannot end at its latest end time. Furthermore, energetic reasoning can easily be extended to discrete resources.

A good starting point to learn more about edge-finding and energetic reasoning on unary resources is [1] where the authors describe and compare several variants of these techniques. Although these tools (edge-finding, energetic reasoning) are very efficient in pure scheduling problems, they suffer from the same limitations as timetabling techniques. Because they consider the absolute position of activities in time rather than their relative position, they will not propagate until the time windows of activities have become small enough and the propagation may be very limited in case the current schedule contains many precedence constraints. Furthermore, these tools are available for unary and discrete resources only and are difficult to generalize to reservoirs.

The following sections of this paper describes two new techniques to propagate discrete and reservoir resources based on analyzing the relative position of activities rather than their absolute position. These algorithms fully exploit the precedence constraints between activities and propagate even when the time windows of activities are still very large which is typically the case in least-commitment planners and schedulers. Of course these new propagation algorithms can be used in cooperation with the existing techniques we just described above. Both of our algorithms are based on the precedence graph structure described in the section below.

4 Precedence Graph

4.1 Definitions

A **resource event** x on a given resource R is a time-point variable at which the availability of the resource changes because of an activity. A resource event always corresponds to the start or end point of an activity. Let:

- $t(x)$ denote the time-point variable of event x. $t_{min}(x)$ and $t_{max}(x)$ will respectively denote the current minimal and maximal value in the domain of $t(x)$.
- $q(x)$ denote the relative change of resource availability due to event x with the convention that $q > 0$ denotes a resource production and $q < 0$ a resource consumption. $q_{min}(x)$ and $q_{max}(x)$ will respectively denote the current minimal and maximal value in the domain of $q(x)$.

There is of course an evident mapping between the resource constraints on a resource and the resource events. Note that all time extents are associated a unique resource event except for $FromStartToEnd$ that is associated two.
A **precedence graph** on a resource R is a directed graph $G_R = (V, E_{\leq}, E_{<})$ where $E_{<} \subset E_{\leq}$ and:

- V is the set of resource events on R
- $E_{\leq} = (x, y)$ is the set of precedence relations between events of the form $t(x) \leq t(y)$.
- $E_{<} = (x, y)$ is the set of precedence relations between events of the form $t(x) < t(y)$.

The precedence graph on a resource aims at collecting all the precedence information between events on the resource. These precedence information may come from: (1) temporal constraints in the initial statement of the problem, (2) temporal constraints between activities in the same planning operator, (3) search decisions (e.g. causal link, promotion, demotion, ordering decisions on resources) or (4) may have been discovered by propagation algorithms (e.g. unsafe causal links handling, disjunctive constraint, edge-finding, etc.) or simply because $t_{max}(x) \leq t_{min}(y)$. When new events or new precedence relations are inserted, the precedence graph incrementally maintains its transitive closure. This leads to a worst-case complexity of $O(n^2)$ to maintain the precedence graph. The precedence relations in the precedence graph as well as the initial temporal constraints are propagated by an arc-consistency algorithm. Given an event x in a precedence graph and assuming the transitive closure has been computed, we define the following subsets of events:

- $S(x)$ is the set of events simultaneous with x that is the events y such that $(x, y) \in E_{\leq}$ and $(y, x) \in E_{\leq}$
- $B(x)$ is the set of events before x that is the events y such that $(y, x) \in E_{<}$

- $BS(x)$ is the set of events before or simultaneous with x that is the events y such that $(y,x) \in E_{\leq}$, $(y,x) \notin E_{<}$ and $(x,y) \notin E_{\leq}$
- $A(x)$ is the set of events after x that is the events y such that $(x,y) \in E_{<}$
- $AS(x)$ is the set of events after or simultaneous with x that is the events y such that that $(x,y) \in E_{\leq}$, $(x,y) \notin E_{<}$ and $(y,x) \notin E_{\leq}$
- $U(x)$ is the set of events unranked with respect to x that is the events y such that $(y,x) \notin E_{\leq}$ and $(x,y) \notin E_{\leq}$

Note that $(S(x), B(x), BS(x), A(x), AS(x), U(x))$ is a partition of V. An example of precedence graph with an illustration of these subsets is given on Figure 1 and corresponds to a schedule with the 6 resource constraints: $< A_1, R, -2, FromStartToEnd >$, $< A_2, R, [-10,-5], AfterStart >$, $< A_3, R, -1, AfterStart >$, $< A_4, R, 2, AfterEnd >$, $< A_5, R, 2, AfterEnd >$, $< A_6, R, 2, AfterEnd >$ and some precedence relations. The subsets are relative to the event x corresponding to the start of activity A_1.

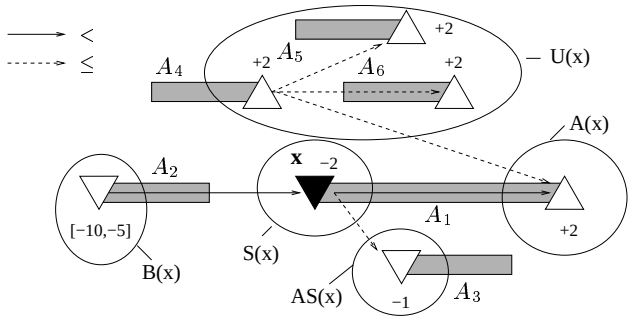


Fig. 1. An Example of Precedence Graph

4.2 Implementation and Complexity

As we will see in next section, our propagation algorithms often need to query the precedence graph about the relative position of two events on a resource so this information needs to be accessible in $O(1)$ on our structure. It explains why we chose to implement the precedence graph as a matrix that stores the relative position of every pair of events. Furthermore, on our structure, the complexity of traversing any subset of events (e.g. $B(x)$ or $U(x)$) is equal to the size of this subset. Note that the precedence graph structure is extensively used in ILOG Scheduler and is not only useful for the algorithms described in this paper. In particular, the precedence graph implementation in ILOG Scheduler allows the user to write his own complex constraints that rely on this graph as for example the one involving alternative resources and transition times described in [10].

5 New Propagation Algorithms

5.1 Energy Precedence Constraint

The **energy precedence constraint** is defined on discrete resources only. As it does not require that the resource be closed, it can be used at any time during the search. The idea is as follows (for simplicity, we assume that all the resource constraints have

a time extent $FromStartToEnd$). Suppose that Q denotes the maximal capacity of the discrete resource over time. If x is a resource event and Ω is a subset of resource constraints that are constrained to execute before x, then the resource must provide enough energy to execute all resource constraints in Ω between the earliest start times of activities of Ω and $t(x)$. More formally:

$$t_{min}(x) \geq \min_{<A,R,q,TE> \in \Omega}(start_{min}(A)) + \sum_{<A,R,q,TE> \in \Omega}(q_{min}(A) * dur_{min}(A))/Q$$

A very simple example of the propagation performed by this constraint is given in Figure 2. If we suppose that the maximal capacity of the discrete resource is 4 and all activities must start after time 0, then by considering $\Omega = \{A_1, A_2, A_3, A_4\}$, we see that event x cannot be executed before time $[0]+[(2*10)+(2*8)+(2*8)+(2*2)]/4 = 14$. Of course, a symmetrical rule can be used to find an upper bound on $t(x)$ by considering the subsets Ω of resource constraints that must execute after x. The same idea as the energy precedence constraint is used in [19] to adjust the time-bounds of activities on different unary resources.

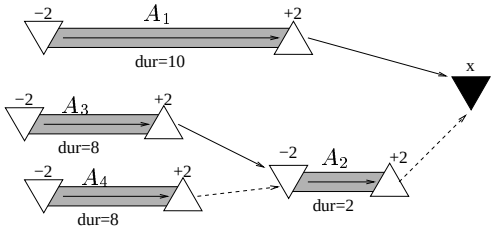


Fig. 2. Example of Energy Precedence Propagation

It's important to note that the energy precedence algorithm propagates even when the time window of activities is very loose (in the example of Figure 2, the latest end times of activities may be very large). This is an important difference with respect to classical energetic and edge-finding techniques that would propagate nothing in this case. The propagation of the energy precedence constraint can be performed for all the events x on a resource and for all the subsets Ω with a total worst-case time complexity of $O(n * (p + log(n))$ where n is the number of the events on the resource and p the maximal number of predecessors of a given event in the graph ($p \leq n$). Note that when the discrete resource has a maximal capacity profile that varies over time, the algorithm can take into account some fake resource constraints with instantiated start and end times to accommodate the maximal capacity profile.

5.2 Balance Constraint

The **balance constraint** is defined on a reservoir resource. When applied to a reservoir, the basic version of this algorithm requires the reservoir to be closed. When applied to a discrete resource, the resource may still be open. The basic idea of the balance constraint is to compute, for each event x in the precedence graph, a lower and an upper bound on the reservoir level just before and just after x. The reader will certainly find some similarities between this constraint and the Modal Truth Criterion on planning predicates first introduced in [6]. Actually this is not surprising as the balance constraint

can be considered as a kind of MTC on reservoirs that only detects some necessary conditions[1]. Given an event x, using the graph we can compute an upper bound on the reservoir level at date $t(x) - \epsilon$ just before x assuming (1) All the production events y that **may** be executed strictly before x are executed strictly before x and produce as much as possible that is $q_{max}(y)$; (2) All the consumption events y that **need** to be executed strictly before x are executed strictly before x and consume as little as possible that is $q_{max}(y)$; and (3) All the consumption events that **may** execute simultaneously or after x are executed simultaneously or after x. More formally, if L_{init} is the initial level of the reservoir, P the set of production events and C the set of consumption events, this upper bound can be computed as follows:

$$L^{<}_{max}(x) = L_{init} + \sum_{y \in P \cap (B(x) \cup BS(x) \cup U(x))} q_{max}(y) + \sum_{y \in C \cap B(x)} q_{max}(y) \quad (1)$$

Applying this formula to event x on Figure 1 if with suppose $L_{init} = 2$ leads to $L^{<}_{max}(x) = 2 + [2 + 2 + 2] + [-5] = 3$. In a very similar way, it is possible to compute $L^{<}_{min}(x)$, a lower bound of the level just before x; $L^{>}_{max}(x)$, an upper bound of the level just after x and $L^{>}_{min}(x)$, a lower bound of the level just after x. For each of these bounds, the balance constraint is able to discover four types of information: **dead ends**, new bounds for **resource usage variables** and **time variables** and new **precedence relations**. For symmetry reasons we only describe the propagation based on $L^{<}_{max}(x)$.

Discovering dead ends. Whenever $L^{<}_{max}(x) < 0$, we know for sure that the level of the reservoir will be negative just before event x so the search has reached a dead end.

Discovering new bounds on resource usage variables. Suppose there exists a consumption event $y \in B(x)$ such that $q_{max}(y) - q_{min}(y) > L^{<}_{max}(x)$. If y would consume a quantity q such that $q_{max}(y) - q > L^{<}_{max}(x)$ then, simply by replacing $q_{max}(y)$ by $q(y)$ in formula (1), we see that the level of the reservoir would be negative just before x. Thus, we can find a better lower bound on $q(y)$ equal to $q_{max}(y) - L^{<}_{max}(x)$. On the example of Figure 1, this propagation would restrict the consumed quantity at the beginning of activity A_2 to $[-8, -5]$ as any value lower than -8 would lead to a dead end.

Discovering new bounds on time variables. Formula (1) can be rewritten as follows:

$$L^{<}_{max}(x) = (L_{init} + \sum_{y \in B(x)} q_{max}(y)) + (\sum_{y \in P \cap (BS(x) \cup U(x))} q_{max}(y))$$

If the first term of this equation is negative, it means that some production events in $BS(x) \cup U(x)$ will have to be executed strictly before x in order to produce at least:

$$\Pi^{<}_{min}(x) = -L_{init} - \sum_{y \in B(x)} q_{max}(y)$$

[1] When the reservoir is not closed, one can imagine extending our propagation algorithm into a real truth criterion on reservoirs that would allow justifying the insertion of new reservoir producers or consumers into the current schedule. This interesting extension clearly worth to study but is out of the scope of this paper.

Let $P(x)$ denote the set production events in $BS(x) \cup U(x)$. We suppose the events $(y_1, \cdots, y_i, \cdots, y_p)$ in $P(x)$ are ordered by increasing minimal time $t_{min}(y)$. Let k be the index in $[1, p]$ such that:

$$\sum_{i=1}^{k-1} q_{max}(y_i) < \Pi^{<}_{min}(x) \leq \sum_{i=1}^{k} q_{max}(y_i)$$

If event x is executed at a date $t(x) \leq t_{min}(y_k)$, not enough producers will be able to execute strictly before x in order to ensure a positive level just before x. Thus, $t_{min}(y_k) + 1$ is a valid lower bound of $t(x)$. On Figure 1 if $L_{init} = 2$, $\Pi^{<}_{min}(x) = 3$, and this propagation will deduce that $t(x)$ must be strictly greater than the minimal between the earliest end time of A_5 and the earliest end time of A_6.

Discovering new precedence relations. There are cases where we can perform an even stronger propagation. Suppose there exists a production event y in $P(x)$ such that:

$$\sum_{z \in P(x) \cap (B(y) \cup BS(y) \cup U(y))} q_{max}(z) < \Pi^{<}_{min}(x)$$

Then, if we had $t(x) \leq t(y)$, we would see that again there is no way to produce $\Pi^{<}_{min}(x)$ before event x as the only events that could eventually produce strictly before event x are the ones in $P(x) \cap (B(y) \cup BS(y) \cup U(y))$. Thus, we can deduce the necessary precedence relation: $t(y) < t(x)$. For example on Figure 1, the balance algorithm would discover that x needs to be executed strictly after the end of A_4. Note that a weaker version of this propagation has been proposed in [4] that runs in $O(n^2)$ and does not analyze the precedence relations between the events of $P(x)$.

Like for timetabling approaches, one can show that the balance algorithm is **sound**, that is, it will detect a dead end on any fully instantiated schedule that violates the reservoir resource constraint. In fact, the balance algorithm does not even need the schedule to be fully instantiated: for example, it will detect a dead end on any non-solution schedule as soon as all the production events are ordered relatively to all the consumption events on a resource. Furthermore, when all events x on a reservoir of capacity Q are so that $L^{<}_{max}(x) \leq Q$, $L^{>}_{max}(x) \leq Q$, $L^{<}_{min}(x) \geq 0$, and $L^{>}_{min}(x) \geq 0$ - in that case, we say that event x is **safe** - then, any order consistent with the current precedence graph satisfies the reservoir constraint. In other words, the reservoir is solved. This very important property allows stopping the search on a reservoir when all the events are safe and even if they are not completely ordered. Note also that, according to the concepts introduced in [14], the balance constraint can be seen as an algorithm that implicitly detects and solves some deterministic MCSs on the reservoir while avoiding the combinatorial explosion of enumerating these MCSs. The balance algorithm can be executed for all the events x with a worst-case complexity in $O(n^2)$ if the propagation that discovers new precedence relations is not turned on, in $O(n^3)$ for a full propagation. In practice, there are many ways to shortcut this worst case and in particular, we noticed that the algorithmic cost of the extra-propagation that discovers new precedence relations was negligible. In our implementation, at each node of the search, the full balance constraint is executed until a fix point is reached.

6 First Results

We implemented a complete and relatively simple search procedure on reservoirs that selects pairs of unsafe events (x, y) and creates a choice point by adding either the relation $t(x) < t(y)$ or $t(y) \geq t(x)$. The heuristics for selecting which pair of events to order relies on the bounds on reservoir levels $L^{<}_{max}(x)$, $L^{>}_{max}(x)$, $L^{<}_{min}(x)$, and $L^{>}_{min}(x)$ computed by the balance constraint. These levels can indeed be considered as some texture measurements [2] projected on the schedule events. Until now, few benchmarks are available on problems involving temporal constraints and reservoirs. The only one we are aware of is [15] where the authors generate 300 project scheduling problems involving 5 reservoirs, min/max delays between activities and minimization of makespan. From these 300 problems, 12 hard instances could not be solved to optimality by their approach. We tested our algorithms on these 12 open problems[2]. The results are summarized on the table below. The size of the problem is the number of activities. The bounds are the best lower and upper bounds of [15]. The times in the table were measured on a HP-UX 9000/785 workstation. We can see that all of the 12 open problems have been closed in less than 1 minute CPU time. Furthermore, our approach produces highly parallel schedules as the balance constraint implements some sufficient conditions for a partial order between events to be a solution. In the optimal solutions there may be up to 10 activities possibly executing in parallel in the partial order.

Problem	**Size**	*Lower bound*	*Upper bound*	**Optimal**	**CPU Time (s)**
#10	50	*92*	*93*	92	0.21
#27	50	*85*	$+\infty$	96	22.18
#82	50	*148*	$+\infty$	no solution	0.15
#6	100	*203*	*223*	211	1.81
#12	100	*192*	*197*	197	1.61
#20	100	*199*	*217*	199	1.54
#30	100	*196*	*218*	204	56.64
#41	100	*330*	*364*	337	1.70
#43	100	*283*	$+\infty$	no solution	53.90
#54	100	*344*	*360*	344	1.26
#58	100	*317*	*326*	317	1.13
#69	100	*335*	$+\infty$	no solution	7.36

We also tested the energy precedence constraint on unary resources. For this purpose, we wrote a very simple least-commitment search procedure[3] based on the precedence graph that orders pairs of activities on a unary resource and aims at finding very good first solutions. We benched this search procedure on 44 famous job-shop problems (namely: *abz5-9*, *ft6*, *ft10*, *orb1-10*, *la1-30*) with the energy precedence constraint as well as the disjunctive and the edge-finder constraint. In average, the makespan of the first solution (without using any restart or randomization) produced by our approach is only 7.35% greater than the optimal makespan whereas the average distance to optimal of the best greedy algorithms so far [17] is 9.33% on the same problems.

7 Conclusion and Future Work

This paper describes two new algorithms for propagating resource constraints on discrete resources and reservoirs. These algorithms strongly exploit the temporal relations in the partial schedule and are able to propagate even if the time windows of activities

[2] All the other problems were easily solved using our approach.
[3] The C++ code of this search procedure is available in the distribution of ILOG Scheduler.

are still very large. Furthermore, on discrete resource, they do not require the resource to be closed. These features explain why they particularly suit integrated approaches to planning and scheduling. From the standpoint of pure scheduling, these algorithms are powerful tools to implement **complete** and **efficient** search procedures based on the relative position of activities. An additional advantage of this approach is that it produces **partially ordered solutions** instead of fully instantiated ones. These solutions are more robust. All the algorithms described in this paper have been implemented and are available in the current version of ILOG Scheduler [12]. As far as AI Planning is concerned, future work will mainly consist in studying the integration of our scheduling framework into a HTN or a POP Planner as well as improving our search procedures.

References

[1] P. Baptiste and C. Lepape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings IJCAI-95*, 1995.

[2] C. Beck, A. Davenport, E. Sitarski, and M. Fox. Texture-based heuristics for scheduling revisited. In *Proceedings AAAI-97*, 1997.

[3] J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. *Annal of Operation Research*, 26:269–287, 1990.

[4] A. Cesta and C. Stella. A time and resource problem for planning architectures. In *Proceedings ECP-97*, 1997.

[5] A. Cesta, A. Oddi, and S. Smith. A constraint-based method for project scheduling with time windows. Technical Report, CMU RI Technical Report, 2000.

[6] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[7] B. Drabble and A. Tate. The use of optimistic and pessimistic resource profiles to inform search in an activity based planner. In *Proceedings AIPS-94*, pages 243–248, 1994.

[8] J. Erschler. *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement*. PhD thesis, Université Paul Sabatier, 1976.

[9] J. Erschler, P. Lopez, and C. Thuriot. Raisonnement temporel sous contraintes de ressources et problèmes d'ordonnancement. *Revue d'IA*, 5(3):7–32, 1991.

[10] F. Focacci, P. Laborie, and W. Nuijten. Solving scheduling problems with setup times and alternative resources. In *Proceedings AIPS-00*, pages 92–101, 2000.

[11] F. Garcia and P. Laborie. *New Directions in AI Planning*, chapter Hierarchisation of the Search Space in Temporal Planning, pages 217–232. IOS Press, Amsterdam, 1996.

[12] ILOG. ILOG Scheduler 5.1 Reference Manual, 2001. http://www.ilog.com/.

[13] S. Khambhampati and X. Yang. On the role of disjunctive representations and constraint propagation in refinement planning. In *Proceedings KR-96*, 1996.

[14] P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings IJCAI-95*, pages 1643–1649, 1995.

[15] K. Neumann and C. Schwindt. Project scheduling with inventory constraints. Technical Report WIOR-572, Universität Karlsruhe, 1999.

[16] W. Nuijten. *Time and resource constrained scheduling: A constraint satisfaction approach*. PhD thesis, Eindhoven University of Technology, 1994.

[17] D. Pacciarelli and A. Mascis. Job-shop scheduling of perishable items. In *Proceedings INFORMS-99*, 1999.

[18] D.E. Smith, J. Frank, and A.K. Jonsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1), 2000.

[19] F. Sourd and W. Nuijten. Multiple-machine lower bounds for shop scheduling problems. *INFORMS Journal of Computing*, 4(12):341–352, 2000.

An Extended Functional Representation in Temporal Planning: towards Continuous Change

Romain Trinquart and Malik Ghallab

LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse, France,
romain.trinquart@laas.fr, malik@laas.fr

Abstract. This paper is concerned with temporal planning relying on CSP-based functional representations. These powerful representations are today mostly restricted to the use of piecewise constant functions ranging over finite domains.
We are proposing here an extension that brings a significant enhancement in the expressiveness of the representation, towards handling continuous change. This extension consists mainly in allowing *piecewise linear functions over continuous domains*. We have studied this extension and we are currently implementing it in the context of the IXTET planner. However this extended representation is not specific to that planner and it can be useful to most temporal planners. We show how IXTET syntax, planning algorithm and control can be simply extended to this class of functions. We then consider the more significant modifications required in the two constraints managers for handling temporal and atemporal CSPs.

1 Introduction

Planning is about time; it is mainly the synthesis of a temporal projection over the future for achieving some desired goal. Time appears to be the main component in a planning ontology. However, *classical planning* has abstracted away time in state transition systems. This restrictive assumption proved to be fruitful in developing planning algorithms and techniques. But its well-known drawbacks in expressiveness - concurrency, duration, persistence, actions for maintaining values, temporally qualified goals and dynamics - and the lack of direct applications of this technology argue more and more for pursuing the development of explicit temporal planning.

Instead of global, still pictures of the entire world, i.e. states, and their transitions, the temporal representation we are interested in here focuses on local individual evolutions of state variables. It is a *functional representation*: each state variable is a partially specified function of time, called a *timeline* or *history*. It is a *CSP-based representation*: these temporal functions are linked through a set of temporal and atemporal constraints into a globally consistent temporal projection.

This potentially powerful representation has been developed and used by several planners, among which IXTET [Laborie 95], RAX-PS
[Muscettola 97] or parcPlan [El-Kholy 96]. But is has been mostly restricted to piecewise constant functions ranging over finite domains. The contribution of this paper is to extend the representation to the use of piecewise linear functions, ranging over continuous or discrete domains.

This representation requires significant extension in the CSP manager of atemporal variables, going from finite domains to mixed finite and continuous domains. It further introduces an interesting coupling between atemporal and temporal variables that were up to now managed separately. This coupling requires another extension of the Time-Map manager.

We studied these extensions and we are currently implementing them in the context of the IxTeT planner. The paper presents the proposed extension within IxTeT notations and construct. However the approach is not specific to that planner, it can be useful to most temporal planners. To be self-contained (and to further promote the CSP-based functional representation) we will describe it in section 2 together with the proposed extension. We will develop in section 3 the required extensions in planning algorithms and control. We then focus on the two atemporal and temporal CSP networks, their extensions and links required by the new type of functions introduced. Algorithms will be described and discussed, however no empirical results on the performance of the extended planner are yet available. Similarly, we'll focus the paper on state variables without considering at this stage the resource handling capabilities of IxTeT. We conclude on the expressiveness of the extended representation and its relationship to other approaches.

2 Representation

In IxTeT, a dynamic domain is described by a set of temporally qualified attributes (multi-valued fluents), each being a k-ary mapping from a finite domain into a finite range. Time is considered as a linearly ordered set of instants. We use time-points as the elementary primitive. Conjunctions of constraints (both symbolic constraints of the time-point algebra and numerical ones) can be expressed between time-points. As will be discussed later, the system does not handle disjunctions. Thus numerical constraints are upper and lower bounds on the temporal distance between two time points.

To describe the dynamic of the world, a propositional reified logic formalism is used, where attributes are temporally qualified by two predicates : *hold* and *event*. The persistence of the value v of an attribute p during the interval *[t,t'[* is expressed by the assertion *hold(p:v,(t,t'))*. The instantaneous change of the value of p from v to v' at time t is expressed as *event(p:(v,v'),t)*.

A partial plan, as well as a planning operator (called a *task*), is represented by a *chronicle*, that is to say a conjunction of temporal predicates (*event* and *hold* predicates) and of constraints on time-points and on atemporal variables used as values or arguments of attributes. The constraints allowed in IxTeT are binary constraints of the form $t \leq t'$ or $t - t' \in [l, u]$ (where t and t' stand for time points) and $x = y$, $x \neq y$, $x \in D$ or $x \in D_x \Rightarrow y \in D_y$ (where x and y stand for atemporal variables).

Chronicles are the core of the representation from the planner's point of view. This representation offers considerable expressiveness by allowing to distribute events and persistence conditions within a task at different time points. Through chronicles, we benefit from a partial specification of the world along some threads caused by actions. Note that an *event* expresses, within a single proposition, both a precondition and an effect of an action : *event(p:(v,v'),t)* requires that p has value v before t and causes p to be equal to v' after t; furthermore this can be expressed anywhere with respect to the start and end points of the action.

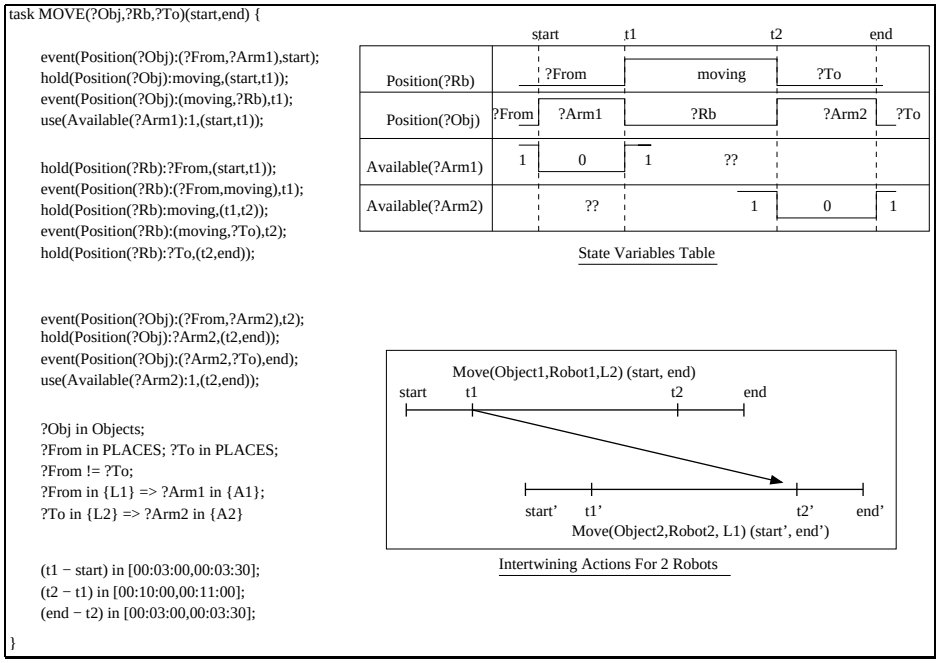


Table 1. A Task Example in the IXTET Formalism

Table 1 presents an example of a task in a simple domain, where a robot can move an object from one place to another. It shows how two instances of this task can be interleaved for two coordinated robots. This instance of a chronicle shows several key points in the representation, such as events occurring at different time points and the possibility for the planner to compute constraints that lead to interleaved actions, thus providing more efficient and flexible plans. We should also point out that tasks variables within a chronicle are not necessarily instantiated but only constrained.

Proposed extension Clearly, a conjunction of *event* and *hold* predicates on some attribute p corresponds to a partially specified function of time ranging over the finite domain of p. An *event* is a step of that function at some time point. A *hold* is a constant value over some time interval. Only piecewise constant function ranging over finite sets can be expressed with *event* and *hold* predicates qualifying attributes over finite domains.

Very few dynamic domains are easily approximated with piecewise constant functions. The position of a robot or the orientation of a satellite do not change along a single step. Admittedly, intermediate positions or orientations are not always of interest at the planning abstraction level. The solution here is to use an unspecified attribute value, e.g. *moving* in the above example, between the values of interest. However, this makes a poor use of the flexibility of the CSP-based functional representation, in particular for interleaving activities and adding incrementally specifications and constraints on attributes while planning. The

moving value cannot be combined with values of other attributes to permit or to forbid other activities at some intermediate positions or orientations. In order to do that, one may break down the change of the corresponding attribute into several steps. However this approach is not very satisfactory; it complicates the specification of planning operators. Furthermore, it can be less efficient than handling directly an extended representation.

We propose here to add to *event* and *hold* a third temporal predicate, called *change*. The expression $change(p : (v, v'), (t, t'))$ specifies that the value of attribute p changes linearly from v to v' during the interval t to t'. Predicate *change*, together with *event* and *hold*, extends the representation to partially specified piecewise linear functions.

The specification of the evolution of p over $[t, t'[$ will be completed by an explicit constraint of the form $v' - v \in D$ or $v' - v = a * (t' - t)$. This last kind of constraint brings about an interesting aspect of the proposed extension since it enables one to connect the effect of an action to its duration which is necessary to express many real world domains.

Having added predicate *change*, the assumption that attributes range over finite domains cannot hold anymore. Consequently, we also added numerical attributes ranging over intervals in the set of real numbers. Constraints concerning numeric variables are : $x \in D$ to specify an initial domain, $x \in D_x \Rightarrow y \in D_y$ to express range dependency (these constraints might mix discrete and numeric variables) and also the two kind of constraints on difference mentioned above.

These two extensions require some slight generalization of the planning and control algorithms, but they mainly require a re-design of the atemporal CSP manager, and, because of the coupling introduced, of the temporal constraint manager as well.

3 Planning Algorithm and control

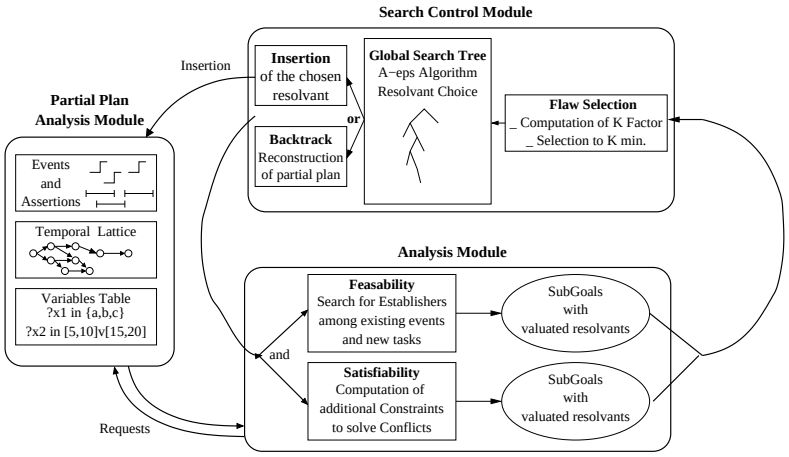


Fig. 1. The Different Modules in IxTeT

IxTeT performs a least commitment search in a space of partial plans [Weld 94]. Nodes of the search space are chronicles. Edges correspond to plan refinements such as adding a task or a constraint. A chronicle Π is a valid solution plan iff:

- (1) For every assertion $hold(Att(x_1,...,x_k) : v,(t,t'))$ and for every event $event(Att(x_1,...,x_k) : (v,v'),t)$ Π contains :
  - an establisher $E = event(Att(x_{est1},...,x_{estk}) : (w,v_{est}),t_{est})$ such that $(x_1 = x_{est1}) \wedge \ ... \ \wedge (x_k = x_{estk}) \wedge (v_{est} = v) \wedge (t_{est} < t)$
  - a causal link $hold(Att(x_1,...,x_k) : v,(t_{est},t))$.
- (2a) For every couple (E,H) where $H = hold(Att(x_1,...,x_k) : v_1,(t_1,t'_1))$ and $E = event(Att(y_1,...,y_k) : (v_2,v'_2),t_2)$ Π contains one of the following constraints : $(t_2 < t_1) \vee (t'_1 < t_2) \vee (x_1 \neq y_1) \vee \ ... \ \vee (x_k \neq y_k)$
- (2b) For every couple (H_1,H_2) where $H_1 = hold(Att(x_1,...,x_k) : v_1,(t_1,t'_1))$ and $H_2 = hold(Att(y_1,...,y_k) : v_2,(t_2,t'_2))$ Π contains one of the following constraints : $(t'_2 < t_1) \vee (t'_1 < t_2) \vee (x_1 \neq y_1) \vee \ ... \ \vee (x_k \neq y_k)$
- (3) The temporal constraints and atemporal constraints are consistent.

This criteria enables us to decide why and how a chronicle should be refined in order to achieve every subgoal (unexplained propositions) through the insertion of tasks and/or causal links, and in order to solve any potential conflict through the insertion of temporal or atemporal constraints.

The first two conditions above are used to detect flaws and to define resolvers. The selection of the next flaw to consider and the non-deterministic choice of a resolver are the main control decisions of the planner. The third condition on the consistency of the constraints does not direct the search process. At each step of the search, constraints are dynamically inserted. They are locally propagated through the corresponding constraint network. If the local propagation detects an inconsistency, then the search process has to backtrack. The local propagation being incomplete for atemporal variables, it is completed by a global consistency checking once a plan is found.

We are going to use the same approach for the extended representation. As an *event*, a *change* is both a precondition and an effect. Hence it may introduce a subgoal requiring an establisher. It can also produce such an establisher. Similarly there can be threats between a *change* and a *hold*, or a between *change* and an *event*, or with another *change*.

Let us consider an unexplained *change* predicate: $change(Att(x_1,...,x_n) : (v_i,v_f),(t_i,t_f))$. It can be established by another such predicate, already in the current chronicle or to be added through some task: $change(Att(x'_1,...,x'_n) : (v'_i,v'_f),(t'_i,t'_f))$ if the following conditions are consistent with the current chronicle:

$x_1 = x'_1 \ and \ ... \ and \ x_n = x'_n \ and \ v'_f = v_i and \ t'_f \leq t_i$.

In that case, the corresponding resolver is the above conjunction together with the following causal link : $hold(Att(x_1,...x_n) : v_i,(t'_f,t_i))$.

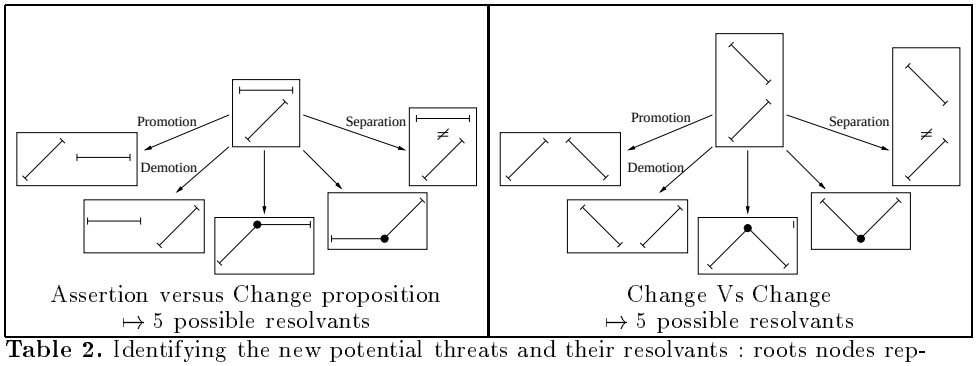
A threat between two predicates: $hold(Att(x_1,...x_n) : v,(t_1,t_2))$ and $change(Att(x'_1,...,x'_n) : (v'_1,v'_2),(t'_1,t'_2))$ can be solved by this disjunction of constraints, when consistent with the current chronicle:

$$\begin{array}{rl} (x_1 \neq x'_1) \ or \ ... \ or \ (x_n \neq x'_n) & \text{(separation)} \\ or \ (t_2 < t'_1) & \text{(promotion)} \\ or \ (t'_2 < t_1) & \text{(demotion)} \\ or \ (t_1 = t'_2 \ and \ v = v'_2) & \\ or \ (t'_1 = t_2 \ and \ v = v'_1) & \end{array}$$

Similarly a threat between $change(Att(x_1, ..., x_n) : (v_1, v_2), (t_1, t_2))$ and $change(Att(x'_1, ..., x'_n) : (v'_1, v'_2), (t'_1, t'_2))$ is solved by this disjunction of constraints:

$$\begin{array}{rl} (x_1 \neq^? x'_1) \ or \ \ldots \ or \ (x_n \neq x'_n) & \text{(separation)} \\ or \ (t_2 < t'_1) & \text{(promotion)} \\ or \ (t'_2 < t_1) & \text{(demotion)} \\ or \ (t_1 = t'_2 \ and \ v_1 = v'_2) & \\ or \ (t'_1 = t_2 \ and \ v_2 = v'_1) & \end{array}$$

The various cases of threats introduced by the extended representation are summarized in the table 2. They can all be handled in a similar way by adding to the current chronicle a conjunction of temporal and atemporal constraints. The remaining problem is how to handle these extended constraints.



Assertion versus Change proposition
$\mapsto$ 5 possible resolvants

Change Vs Change
$\mapsto$ 5 possible resolvants

Table 2. Identifying the new potential threats and their resolvants : roots nodes represent potential conflicts, branches correspond to possible resolvers.

4 The Constraints Managers

IxTeT has three different specialized constraints managers : one dealing with atemporal variables, one dealing with time points and a third one dealing with resources. In order to cope with the new kind of constraints needed to handle the extension to continuous change, we need to enhance the atemporal constraints manager and to bring about closer interactions between this manager and the Time-Map.

4.1 The Atemporal Constraints Manager

This system handles the atemporal variables which are dynamically introduced in the partial plan during the search process, as well as the related constraints which might be introduced to solve flaws. It is actually a CSP Solver dealing with variables over finite domains.

The manager is queried by the planning system for testing if two variables can be unified or differentiated, for inserting new constraints and for checking

global consistency. The desired level of expressivity required by the planner is quite rich. We have to handle the following constraints : $x = y$, $x \neq y$, $x \in D$, $x \in D_x \Rightarrow y \in D_y$.

Thus the manager problem is that of a general binary CSP over finite domains whose consistency checking is a NP-complete problem. Because of the large number of queries from the planner to the manager, IxTeT maintains an incomplete arc-consistency checking while planning. Complete consistency checking is postponed until a plan without flaws is found.

The main drawback of this method is that it leads to "blind" backtrack : the search process might choose to develop a refinement branch that is doomed to fail because of an undetected inconsistency, thus loosing time. Furthermore, once the inconsistency is detected the backtrack point cannot be easily identified. However this criticism should be counterbalanced by the following observation : because of the least commitment strategy, carrying on the search process will lead to a reduction of the problem size by introducing equality constraints corresponding to postponed establishment.

Dealing with continuous domains : To handle *change* predicates, the planner needs to get information concerning variables whose domains are continuous. The constraints that might be posted on two numeric variables are : $x \in D$, $x - y \in D'$ and $x \in D_x \Rightarrow y \in D_y$.

This last class of constraints leads us to consider domains that are disjunctions of intervals in $\mathbb{R}$: given a constraint "$x \in D_x \Rightarrow y \in D_y$", if $Dom(y) \cap D_y = \emptyset$ then the propagation yields the insertion of the constraint $x \notin D_x$ which might create "holes" in the domain associated to x.

The local arc consistency algorithm can be adapted without too much trouble. We have to provide operators on the intervals disjunctions similar to those used for discrete set and also to define propagation rules concerning the new "difference" constraints.

Problems arise concerning the complete propagation algorithm. This algorithm performs an extensive instantiation of the variables to detect the values that will necessary lead to inconsistency. It is a forward checking search . This cannot be directly transposed to numeric variables : it makes no sense to check for all the values in a continuous domain. A naive technique consists in replacing variable instantiations by domain splitting. A dichotomy splitting can be iterated until a fixed minimal interval length is reached. We propose to use an analog method, but instead of a systematic dichotomy that does not take the constraint network into account and thus might lead to unnecessary computations, we will use the constraints to guide the splitting of intervals.

Our purpose is to have a common framework to handle both discrete and numeric variables. To achieve this goal we have got to change the notion of instantiation concerning numeric variables so as to obtain equivalent complexity. Let us point out that , in the discrete case, instantiations are performed in order to reduce the uncertainty on the value of a variable, thus enabling the solver to actually fire every constraint. But, hopefully, the triggering of the constraints concerning numeric variables does not require the variables to be fully instantiated. Among the constraints on numeric variables, only the dependency constraints propagation might be postponed because of a lack of information. The equality and domain restriction constraints will be immediately propagated

at insertion, respectively through the fusion of the equivalence class of the affected variables and through the effective reduction of the valuation domain. The distance constraints ($x - y \in D$) will be propagated by the arc consistency propagation. But to propagate a dependency constraint "$x \in D_x \Rightarrow y \in D_y$", the solver needs to know whether x (resp. y) takes its value in D_x (resp. D_y) or not. This is the kind of constraints that might stop propagation along paths and thus prevent the detection of inconsistency. The complete propagation algorithm should reduce the uncertainty on the variables enough to decide how to propagate the constraint, i.e. it should decide whether $x \in D_x$ or not. This is the key to our algorithm.

The domain of a numeric variable x will be decomposed into disjoint intervals I_k such that for every constraint "$x \in D_x \Rightarrow y \in D_y$" or "$z \in D_z \Rightarrow x \in D_x$", the following proposition is true : $(I_k \subset D_x) or (I_k \cap D_x = \emptyset)$. The complete propagation procedure will then successively try to reduce the valuation domain of x to these different intervals (I_k). For each I_k it will be able to decide how the constraints should be applied. Thus the problem of instantiating a numeric variable has been transformed into choosing subintervals in a finite set.

Since this method can also be used for discrete variables (domains will then be decomposed into disjoint subsets), it provides a simple algorithm to deal with a heterogeneous CSP containing both discrete and numeric variables for the type of constraints in a temporal planner. Table 3 presents the algorithm for complete propagation in such a CSP. The procedure Complete_Propagation computes minimal domains in the CSP. For every variable, it builds a partition of its domain according to the constraints. Then it successively shrinks the domain to every element of the partition and checks the consistency of the resulting CSP : only sub-domains leading to consistent CSP are kept in the minimal domain. If a minimal domain is found to be empty, then the CSP is inconsistent.

4.2 Mixing timePoints and Variables

Among the extensions of the representation presented above, we have insisted on the necessity to express actions whose effects depend on their duration. It implies to be able to handle "transversal" constraints binding both timepoints and atemporal variables, i.e. coupling constraints between the atemporal CSP and the Time-Map.

To illustrate such a constraint let us consider a simple example dealing with the control of a robot : an action "Move" brings a robot from one point to another along an axis at constant speed. Thus the duration of the shift is proportional to the distance between the initial and final position of the robot : $Pos_f - Pos_i = speed * (t_f - t_i)$.

The first observation is that this kind of constraint can be splitted, each part being inserted in a specific Constraints Manager : the difference T between t_f and t_i can be introduced in the atemporal variables manager trough the constraints ($Pos_f - Pos_i = speed * T$) and the other way round, the difference D between Pos_f and Pos_i yields the constraints ($t_f - t_i = D/speed$) to be added to the Time-Map. Since both the Time-Map and the atemporal constraints network are dynamically constructed, we shall iterate such constraints posting during the search process. It should be noted however that the resolution of constraints problems computes minimal valuation domain. Thus from one step to another, the domain associated to a variable will shrink : consequently if two distance

Table 3. The Complete Propagation Algorithm

```
Complete_Propagation(){
   \*** computes minimal domains and ***\
   \*** returns true if the CSP is consistent ***\
   For each variable var
      AcceptedValues ← ∅
      Elts ← Partition(var)
      For each e in Elts
         If (unify(var, e) ∧ Global_consistency_checking())
            AcceptedValues.append(e)
         endIf
      endFor
      If AcceptedValues = ∅
         RETURN False
      Else
         Domain(var) ← AcceptedValues
      endIf
   endFor
   RETURN True
}
```

```
Global_Consistency_Checking(){
   \*** returns true if the CSP is consistent ***\
   var ← Choose One Non Instanciated variable
   Values ← Partition(var)
   While Values ≠ ∅
      elt ← pop(Values)
      If (unify(var, elt) ∧ Global_consistency_checking())
         \*** unify triggers local propagation ***\
         RETURN True
      endIf
   endWhile
   RETURN False
}
```

```
Partition(v){
   \*** This function returns the set of alternative ***\
   \*** disjoint domains for v ***\
   If there is at least one ≠ constraint on v then
      RETURN {{e}, e ∈ Domain(v)}
   Else
      Part ← {Domain(v)}
      LDep ← {D, ∃(v', D'), (v ∈ D ⇒ v' ∈ D') ∨ (v' ∈ D' ⇒ v ∈ D)}
      While LDep ≠ ∅
         KDom ← pop(LDep)
         NewPart ← ∅
         While (Part ≠ ∅)
            Elt ← pop(Part)
            NewPart ← NewPart ∪ (Elt ∩ KDom)
            NewPart ← Newpart ∪ (Elt ∩ ∁_KDom)
         endWhile
         Part ← NewPart
      endWhile
      RETURN Part
   endIf
}
```

constraints on the same variable are successively posted, the second one will discard its predecessors.

The second point we must focus on is the difference in handling disjunctions by the two constraints managers. As mentioned in the previous paragraph, domains associated to numeric atemporal variables are disjunctions of intervals. On the contrary the use of disjunctions in the time-map manager is forbidden. To bridge the gap between these two managers we will discuss more in-depth the time-map manager.

Basically this manager handles Simple Temporal Networks [Dechter 89]. It has limited expressivity but complete propagation can be computed in polynomial time thanks to the Floyd-Warshall algorithm, for instance. On the contrary, if we extend the expressivity to disjunctions of intervals (that is to say Temporal CSP or T-CSP), we are then confronted to NP-Hard problems [van Beek 89]. The solution retained in IXTET is to limit the expressivity so as to ensure completeness at every step of the planning process and to deal with disjunctions at the control level with explicit estimates and heuristics.

A possible solution to the coupling between the two CSPs would be to keep simple intervals inside of the temporal network, and to relax constraints inherited from the atemporal variables manager by approximating a disjunction of intervals to its lower and upper bounds. The propagation would then be incomplete because of the relaxation : an inconsistency may be detected only once the time-map manager is committed to instantiate a variable to a value that was not part of the initial disjunction on the atemporal manager's side. The handling of such a situation will be achieved at the search level that should backtrack in order to avoid the inconsistent constraint.

Another solution is to accept to pay the cost of handling disjunctions of intervals in the temporal constraint network. This would enable the search process to rely on complete propagation between the two constraints managers and to backtrack immediately if a constraint is introduced that gives rise to inconsistency.

We have chosen this second solution. The use of disjunctive constraints has long been avoided because it was known to be intractable in worse case. But recent experiment results prove that in most practical cases actual complexity of propagation of sets of intervals through arithmetic constraints such as the one we are concerned with is linear rather than exponential [Shapiro 99]. Furthermore this simplifies the control process, gets rid of heuristic evaluation and avoid backtracking, which our experiments proved to be a real bottleneck for the planner.

5 Discussion and conclusion

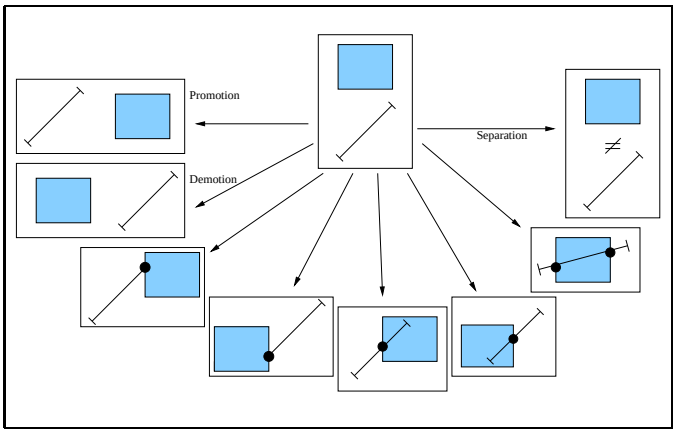
Most of the temporal planners that address the same goals as IXTET rely on the same kind of representation of the world. The most spectacular application in the field was the Remote Agent Experiment lead by the NASA : among the different technologies that were embedded in the remote agent architecture, the planning system PS had a crucial role. In RAX-PS, the world is also described through a set of temporally qualified state variables, some of which represent activities executed by a device : there is a strong parallel between a token associated to a literal and the invocation of an activity to be executed by the execution supervisor. A time point represents the moment at which the activity corresponding

to the new token starts executing. The representation in RAX-PS differs from the one in IxTeT on the following aspects. First in RAX-PS there is no explicit representation of events. The evolution of a variable (time line) is represented by a succession of temporal assertion (*hold*). This leads to slight differences in the search process since it is then guided by completely filling time-lines with assertions, the consistency between two successive assertions being defined by "compatibility rules". This brings out the second difference : in RAX-PS, there are no planning operators such as the *Tasks* mentioned above. The system is only concerned with attribute's value (which represent real procedures for the executive system) and benefits from rules to establish links between assertions along different time lines. Beyond those differences, RAX-PS relies, as IxTeT does, on a set of piecewise constant functions and on a CSP-based approach. It can benefit from our proposed extension. [1]

Among temporal planners, parcPLAN presents similar expressivity, and seems to offer quite interesting performance [Liatsos 99]. It relies on relatively similar representation and search strategy (constraint-based and least commitment). The main difference with IxTeT is the strong separation in the search process between goal achievement and scheduling reasoning. Here again piecewise linear functions could be beneficial.

Last but not least, [Penberthy 94] presented ZENO, a temporal planning system that offered rich expressivity to describe the dynamic of the world. Metric preconditions and effects as well as linear evolution were supported. However these features were supported without the benefit of constraints network methods that enable the search control to concentrate on establishing constraints instead of selecting exact values for variables. It should be noted that, as far as we know, Zeno has only been tested on simple toy domains.

Table 4. Threats concerning the extended hold predicate



[1] Following an interesting remark from an anonymous referee, we checked out in the available literature to what extent the temporal features described in this paper are supported in RAX-PS and we found no clear indication that this system handles temporal primitive other than piece-wise constant values.

An implementation of the extension presented in this paper is currently under process within IxTeT. We plan to compare it to the technique that approximates a *change* through multiple steps in complex domains. Another extension we are considering is a relaxed *hold* predicate : $hold(p : (v, v'), (t, t'))$ specifies that p remains bounded within interval [v,v'] during [t,t'[. Such a predicate is needed to provide better control over continuous evolutions : it enables one to express conditions that might be verified during an evolution, thus increasing possibilities to interleaved actions. The search control could be developed in the same way as for *change* predicate in order to cope with these new assertions. Once again the difficulty is pushed on the constraints solver. Table 4 sums up the possible situations to be examined to deal with such predicates.

References

[Chapman 87] D. Chapman. *Planning for Conjunctive Goals.* Artificial Intelligence, vol. 32, 1987.

[Dechter 89] Rina Dechter, Itay Meiri & Judea Pearl. *Temporal Constraint Networks.* In KR'89: Principles of Knowledge Representation and Reasoning, pages 83–93. Morgan Kaufmann, 1989.

[El-Kholy 96] A. El-Kholy & B. Richards. *Temporal and Resource Reasoning in Planning: the parcPlan approach.* In Proc. ECAI-96., 1996.

[Ghallab 94] M. Ghallab & H. Laruelle. *Representation and Control in IxTeT, a Temporal Planner.* In Proceedings of the International Conference on AI Planning Systems, pages 61–67, 1994.

[Kondrak 97] G. Kondrak & P. Van Beek. *A theoretical evaluation of selected backtracking algorithms.* Journal of AI, vol. 89, pages 365–387, 1997.

[Laborie 95] P. Laborie & M. Ghallab. *IxTeT: an Integrated Approach for Plan Generation and Scheduling.* In Proceedings ETFA-95, 1995.

[Liatsos 97] V. Liatsos & B. Richards. *Least commitment—an optimal planning strategy.* In Proceedings of the 16th Workshop of the UK Planning and Scheduling Special Interest Group, 1997.

[Liatsos 99] Vassilis Liatsos & Barry Richards. *Scaleability in Planning.* In Proc. ECP-99, pages 49–61, 1999.

[Muscettola 97] N. Muscettola, B. Smith, S. Chien, C. Fry, G. Rabideau, K. Rajan & D. Yan. *Onboard Planning for Autonomous Spacecraft.* In Proceedings of the 4th International Symposium on Artificial Intelligence, Robotics and Automation for Space (i-SAIRAS 97), 1997.

[Penberthy 94] J. Scott Penberthy & Daniel S. Weld. *Temporal Planning with Continuous Change.* In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), vol. 2, pages 1010–1015. AAAI Press/MIT Press, 1994.

[Shapiro 99] Rony Shapiro, Yishai A. Feldman & Rina Dechter. *On the Complexity of Interval-Based Constraint Networks.* In MISC'99 Workshop on Applications of Interval Analysis to Systems and Control, 1999.

[van Beek 89] Peter van Beek. *Approximation Algorithms for Temporal Reasoning.* In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pages 1291–1296, 1989.

[Weld 94] Daniel S. Weld. *An Introduction to Least Commitment Planning.* AI Magazine, vol. 15, no. 4, pages 27–61, 1994.

Constraint-based Strategies for the Disjunctive Temporal Problem: Some New Results

Angelo Oddi

IP-CNR, Italian National Research Council
Viale Marx 15, I-00137 Rome, Italy
oddi@ip.rm.cnr.it

Abstract. The Disjunctive Temporal Problem (DTP) involves the satisfaction of a set of constraints represented by disjunctive formulas of the form $x_1 - y_1 \leq r_1 \vee x_2 - y_2 \leq r_2 \vee \ldots \vee x_k - y_k \leq r_k$. DTP is a quite general temporal reasoning problem which includes the well-known Temporal Constraint Satisfaction Problem (TCSP) introduced by Dechter, Meiri and Pearl. This paper describes a basic constraint satisfaction algorithm where several aspects of the current literature are integrated, in particular the so-called *incremental forward checking*. Hence, two new extended solving strategies are proposed and experimentally evaluated. The new results are both very competitive with respect to the current best results and open further research directions that concerns, in particular, the use of *arc-consistency* filtering strategies.

Keywords: constraint reasoning for planning and scheduling, temporal reasoning, constraint algorithms.

1 Introduction

In many recent Artificial Intelligence applications the need of a more expressive temporal reasoning frame is increasing more and more. For example, in continual planning applications [1] a relevant capability is the continuous management of temporal plans [2, 3]. To this purpose, the representation of temporal disjunction allows a leverage of systems' capability, for example, it avoids a too early commitment on action orderings. Other interesting applications of the temporal model proposed in this paper are discussed in the work [4], these applications range from scheduling and planning to temporal database with indefinite information.

The Temporal Constraint Satisfaction Problem (TCSP) [5] is a way to represent temporal disjuctions, it allows constraints of the form $x-y \leq r_1 \vee x-y \leq r_2 \vee \ldots \vee x-y \leq r_k$. A further generalization of the TCSP is proposed in [6, 4] where a problem has constraints of the form $x_1-y_1 \leq r_1 \vee x_2-y_2 \leq r_2 \vee \ldots \vee x_k-y_k \leq r_k$. In [7] this last problem is referred to as Disjunctive Temporal Problem (DTP) and we use this name in the paper. DTPs have been studied in several previous works: (a) in [6, 4] several constraint-based (CSP) algorithms (in the line of [8]) are defined and experimentally compared. One of them, based on *forward checking* [9], is shown to be the best; (b) in [7] DTP is modeled as a propositional satisfability (SAT [10]) problem and solved with a state-of-the-art

SAT-solver plus some additional processing. Experiments show an improvement of up to two orders of magnitude with respect to the results in [6]. (c) finally in [11] the constraint-based algorithm proposed in [6, 4] is improved exploiting the quantitative temporal information in the solution "distance graph". Using this knowledge an incremental version of the forward checking is obtained and shown to be competitive with the results proposed in [7].

Our starting point in the work [11] was the observation of the sharp difference between the results shown in [6] and [7] and the idea of using the "quantitative reasoning" that can come out from a temporal constraint network representation. We basically observe that the effects of quantitative temporal information to improve global performance of DTPs has not been explored enough in previous works. Using such knowledge we were able to define an *incremental forward checking* algorithm which has comparable performance (measured as number of forward checks) with the best SAT-based version proposed in [7].

This paper follows the same CSP-based approach and again focus on using "quantitative reasoning" that can come out from a temporal constraint network representation. In particular we propose a new heuristic strategy for *variable ordering* used in our CSP framework and an *arc-consistency* filtering algorithm. The rationale behind the new results is quite general and can be exploited in other solvers that rely on "quantitative temporal reasoning".

The paper is structured as follows. Section 2 introduces the basic concepts used in the paper. Section 3 gives a basic CSP algorithm which integrates results from both previous works, included the *incremental forward checking*. Section 4 introduces two additional solving algorithms for solving DTP instances, and an experimental evaluation of the different approaches is given in Section 5. Section 6 ends the paper with some conclusions.

2 Preliminaries

The Disjunctive Temporal Problem (DTP) involves a finite set of temporal variables $x_1, y_1, x_2, y_2 \ldots x_n, y_n$ ranging over the reals and a finite set of constraints $C = \{c_1, c_2, \ldots c_m\}$ of the form $x_1 - y_1 \leq r_1 \vee x_2 - y_2 \leq r_2 \vee \ldots \vee x_k - y_k \leq r_k$, where r_i are real numbers. A DTP is consistent if an assignment to the variables exists such that in each constraints $c_i \in C$ at least one disjunct $x_{ij} - y_{ij} \leq r_{ij}$ is satisfied. One way to check for consistency of a DTP consists of choosing one disjunct for each constraint c_i and see if the conjunction of the chosen disjuncts is consistent. It is worth observing that this is equivalent to extracting a "particular" STP (the Simple Temporal Problem defined in [5]) from the DTP and checking consistency of such a STP. If the STP is not consistent another one is selected, and so on. Both previous approaches to DTP [6, 4, 7, 11] do this basic search step.

Previous Work. All [6, 4, 7, 11] share a "two layered" algorithmic structure. An upper layer of reasoning is responsible for guiding the search that extracts the set of disjuncts, a lower layer represents the quantitative information of the temporal reasoning problem. In [6] a general CSP formulation is used at the upper level while the quantitative information is managed by using the incremental di-

rectional path consistency (IDPC) algorithm of [12]. In [7] at the upper level the DTP is encoded as a SAT problem, a SAT-solver extracts an STP to be checked, a simplified version of the Simplex algorithm is used at the lower level to check for its consistency. Stergiou and Kubarakis define different backtracking algorithms for managing the upper-level and experimentally verify that the version using forward checking is the best. Forward checking is used after each choice to test which of the possible next choices are compatible with current partial STP. In the rest of the paper their best algorithm is called SK. Armando, Castellini and Giunchiglia focus their attention on the SAT encoding, each disjunct is a propositional formula, and they use a state of the art SAT-solver enriched with a form of forward checking biased by the temporal information. Their basic version is called $TSAT$ and is shown to improve up to one order of magnitude with regard to SK. Then they add a further preprocessing step called IS that basically produces a more accurate SAT encoding because it codifies mutual exclusion conditions among propositions that exists in the temporal information, but were lost by the first standard encoding.

DTP Consistency Checking as a Meta-CSP. Before introducing our algorithm we underscore the possibility of representing the consistency checking problem as a *meta*-CSP problem, where each DTP constraint $c \in C$ represents a (meta) variable and the set of disjuncts represents variable's domain values $D_c = \{\delta_1, \delta_2, \ldots \delta_k\}$. A *meta*-CSP problem is consistent if exists at least an element S (solution) of the set $D_1 \times D_2 \times \ldots \times D_m$ such that the corresponding set of disjuncts $S = \{\delta_1, \delta_2, \ldots \delta_m\}$ $\delta_i \in D_i$ is temporally consistent.

Each value $\delta_i \in D_i$ represents an inequality of the form $x_i - y_i \leq r_i$ and a solution S can be represented as a labeled graph $G_d(V_S, E_S)$ called "distance graph" [5]. The set of nodes V_S coincides with the set of DTP variables $x_1, y_1, x_2, y_2 \ldots x_n, y_n$ and each disjunct $x_i - y_i \leq r_i$ is represented by a direct edge (y_i, x_i) from y_i to x_i labeled with r_i. A path from a node x_i to y_j on the graph is a set of contiguous edges $(x_i, y_i), (y_i, y_{i1}), (y_{i1}, y_{i2}), \ldots, (x_{il}, y_j)$ and the length of the path is the sum of the edges' labels. The set of disjuncts S corresponds to an STP. S is a solution to the *meta*-CSP problem if G_d does not contain closed path with negative length (*negative cycles*) [5]. From the graph G_d a numerical solution of the problem can be extracted as follows. Let $d_{x_i y_i}$ be the shortest path distance on G_d from the node x_i to y_i, without loss of generality we can assume a variable x_i as reference point, for example x_1, in this way the tuple $(d_{x_1 x_1}, d_{x_1 x_2}, \ldots d_{x_1 x_n})$ is a solution of the original DTP problem. In fact, the previous values represent the shortest distance from the reference node x_1 to all the other ones (in particular $d_{x_1 x_1} = 0$). For each edge $x_i - y_i \leq r_i$ in G_d as it is well known values $(d_{x_1 x_1}, d_{x_1 x_2}, \ldots d_{x_1 x_n})$ must hold the Bellman's inequalities: $d_{x_1 x_i} \leq d_{x_1 y_i} + r_i$, that is $d_{x_1 x_i} - d_{x_1 y_i} \leq r_i$. Hence $(d_{x_1 x_1}, d_{x_1 x_2}, \ldots d_{x_1 x_n})$ is a solution for the DTP.

This view of the consistency checking problem is used to define our CSP approach and in particular is useful to understand our incremental forward checking method.

```
CSP-DTP-SOLVER(dtp, S)
1.  if CheckConsistency(dtp)
2.     then if IsaSolution(dtp)
3.             then return(S)
4.             else begin
5.               c ← SelectVariable(dtp)
6.               δ ← Choose Value(dtp, c)
7.               CSP-DTP-SOLVER(dtp, S ∪ {δ})
8.             end
9.     else return(Fail)
10. end
```

Fig. 1. A CSP solver for the DTP.

3 A CSP Algorithm for DTP

In this work we mainly follow the constraint-based approach of Stergiu and Kubarakis [6, 4] for solving DTP instances. Figure 1 shows a CSP procedure which starts from an empty solution S and basically executes three steps: (a) the current partial solution is checked for consistency (Step 1) by the function *Check-Consistency*. This function filters also the search space from inconsistent states. If the partial solution is a complete solution (Step 2) the algorithm exits. If the solution is still incomplete the following two steps are executed; (b) a (meta) variable (a constraint c_i) is selected at Step 5 by a variable ordering heuristic; (c) a disjunct $x_i - y_i \leq r_i$ is chosen (Step 6) from the domain variable D_i and added to S (represented at the lower level as a G_d graph). Hence the solver is recursively called on the partial updated solution $S \cup \{\delta\}$.

The *CheckConsistency* function is the core of the CSP algorithm, it both updates the set of distances $d_{x_i y_j}$ and the domain variables D_i by forward checking. In particular it executes two main steps:

Temporal propagation. every time a new inequality $x_i - y_i \leq r_i$ is added to the G_d graph, the set of distances $d_{x_i x_j}$ is updated by a simple $O(n^2)$ algorithm.

Forward checking. After the previous step, for each not assigned meta-variable the domain D_i is checked for consistency (forward checking). Given the current solution represented by G_d, each value $x_i - y_i \leq r_i$ belonging to a not assigned variable and which induces a negative cycles on G_d is removed. In other words, each time a value $\delta_i \equiv x_i - y_i \leq r_i$ satisfies the test $r_i + d_{x_i y_i} < 0$, then δ_i is removed from the corresponding domain D_i. In the case that one domain D_i becomes empty, the function *CheckConsistency* returns *false*.

The *CheckConsistency* step contributes to avoid investigation of search states proved inconsistent and the other two steps (Steps 5 and 6 of Figure 1) are used to guide the search according to heuristic estimators.

SelectVariable. It applies the simple and effective Minimum Remaining Values (MRV) heuristic: variables with the minimum number of values are selected first. It is worth noting that the heuristic just ranks the possible choices

deciding which one to do first but all the choices should be done (it is not a non deterministic search step).

ChooseValue. This represents a non deterministic operator which starts a different computation for each domain values. Obviously in our implementation we use a *depth-first* search strategy, where there is no particular values ordering heuristic. However, in the case a constraint (variable) is always satisfied by the current partial solution S_p, that is, a constraint disjunct $x_i - y_i \leq r_i$ exists such that holds the condition $d_{y_i x_i} < r_i$, no branching is created. In fact, the current constraint is implicitly "contained" in the partial solution and it will be satisfied in all the solution created from S_p.

3.1 Integrating SAT Features

The current version of our CSP solver integrates also the so-called *semantic branching* [7]. This is a feature that in the SAT approach comes for free and that in the CSP temporal representation is to be explicitly inserted. It avoids to test again certain conditions previously proved inconsistent. The idea behind semantic branching is the following, let us suppose that the algorithm builds a partial solution $S_p = \{\delta_1, \delta_2, \ldots \delta_p\}$ and a not assigned meta-variable is selected which has a disjunct set of two elements $\{\delta^{'}, \delta^{''}\}$. Let us suppose that the disjunct $\delta^{'}$ is selected first and no feasible solution exists from the partial solution $S_p \cup \{\delta^{'}\}$. In other words, each search path from the node $S_p \cup \{\delta^{'}\}$ arrives to an infeasible state. In this case the depth-first search process removes the decision $\delta^{'}$ from the current solution and tries the other one ($\delta^{''}$). However, even if the previous computation is not able to find a solution, it demonstrates that with regard to the partial solution S_p no solution can contain the disjunct $\delta^{'}$. If we simply try $\delta^{''}$ we lose the previous information, hence, before trying $\delta^{''}$, we add the condition $\neg\delta^{'}$ (that is $x^{'} - y^{'} > r_i$) to the partial solution. It is worth nothing that in this case it is important to make explicit the semantic branching by adding the negation because the values in the domains D_i are not self-exclusive. In other cases, for example a scheduling problem, where branching is done with regard to the temporal ordering of pairs of activities A and B, semantic branching is not useful. In fact when $A\ before\ B$ is chosen the case $B\ before\ A$ is implicitly excluded.

In this section we have described our basic algorithm that integrates some of the previous analysis in a meta-CSP search framework. From now on we call this algorithm CSP and it is the base for the description of the incremental forward checking of the next section.

3.2 Incremental Forward Checking

The algorithms for solving DTP introduced at the beginning of this section is based on the *meta*-CSP schema with some additional features. In particular, it uses the enriched backtracking schema called *semantic branching*. To further improve the performance of the CSP approach we have investigated aspects connected to the quantitative temporal information. This aspect has received less attention in [6, 7, 4]. In particular in this section we introduce a method to

significantly decrease the number of forward checks by using the temporal information. Its general idea is relatively simple.

Rationale. When a new disjunct δ is added to a partial solution, the temporal propagation algorithm inside *CheckConsistency* updates only a subset of the distances $d_{x_i x_j}$ (usually a "small" subset). The forward checking test on disjuncts is performed w.r.t. the distances in the graph G_d. It is of no use to perform a forward checking test of the form $d_{x_i y_i} + r_i < 0$ on a disjunct δ_i when the distance $d_{x_i y_i}$ has not been changed w.r.t. the previous state.

This basic observation can be nicely integrated in CSP with the additional cost of a static preprocessing needed to create for each pair of nodes $\langle x_i, y_j \rangle$ the set of *affected meta values* $AMV(x_i, y_j)$.

Affected meta-values w.r.t. a pair $\langle x_i, y_j \rangle$. Given a distance $d_{x_i y_j}$ on G_d the set of *affected meta values* discriminates which subset of disjuncts are affected by an update of $d_{x_i y_j}$. The set $AMV(x_i, y_j)$ associated to the distance $d_{x_i y_j}$ (or the pair $\langle x_i, y_j \rangle$) is defined as the set of disjuncts $x - y \leq r$ whose temporal variables x and y respectively coincide with the variables y_j and x_i ($AMV(x_i, y_j) \doteq \{x - y \leq r : x = y_j, y = x_i\}$).

Given a DTP, the set of its AMVs is computed once for all with a preprocessing step with a space complexity $O(m + n^2)$ and a time complexity $O(n^3 \ln n)$ (as explained below each set AMV is represented as a sorted list according to the values r). The information stored in the AMVs can be used in a new version of CSP we call "incremental forward checking" ($CSPi$). It requires a modification of the *CheckConsistency* function. The new incremental version of the *CheckConsistency* works in two main steps:

1. The distances $d_{x_i y_j}$ are updated and the set of distances that have been changed is collected.
2. given such set, for each $d_{x_i y_j}$ the corresponding $AMV(x_i, y_j)$ is taken, and its values are forward checked. In particular, all the set $AMV(x_i, y_j)$ are represented as a list of disjuncts sorted according to the value of r and the forward checking test $d_{x_i y_j} + r < 0$ is performed from the disjunct with the smallest value of r. In this way, when a test fails on the list element δ, it will fail also on the rest of the list and the forward checking procedure can stop on $AMV(x_i, y_j)$.

In the experimental section we show that the algorithm $CSPi$ (constraint-based solver with incremental forward checking) strongly improves with respect to the basic CSP and becomes competitive with the best results available in the literature.

4 New Constraint-based Solving Strategies for DTP

In this section we propose two additional solving strategies for DTP based on the work [11]. In particular we propose: (1) a new *variable ordering* heuristic; (2) an *arc-consistency* filtering strategy.

The rational behind the first method is based on the observation that given a DTP problem, and considerd a value $\delta_i \equiv x_i - y_i \leq r_i$, during the solving process δ_i is removed by forward checking from its domain D_i when induces negative cycles in the current solution represented by the G_d graph. On the basis of the previous observation we propose the following variable ordering strategy: *select the subset of variables with minimum number of remaining values $x_i - y_i \leq r_i$ and within this subset, the variable with maximal number of negative coefficients r_i.* The values δ_i with negative coefficients r_i are crucial to the existence of a solution to a DTP. In fact, it is simple to see that a DTP instance without negative r_i values has always a solution. On the other hand, the presence of negative r_i values generate negative cycles on the graph G_d and induces inconsistent partial solutions. This strategy has the main purpose of pruning the search tree in its early stages, trying to create as many as possible negative cycles, in this way the strategy maximizes the probability of finding negative cycles at the early steps of the search tree. As we will see in the experimental section this strategy is effective in the *transistion phase* of a DTP problem where the probability of find a solution is very low.

The second solving method can be explained by giving a new version of the *CheckConsistency* algorithm used in the general algorithmic template described in Figure 1. The aim of this solving method is reducing the dimension of the search tree by the application of a more effective filtering strategy and to explore the possibility of finding tradeoffs among number of consistency checks, number of visited search nodes, and CPU time. In particular, we propose an *arc-consistency* filtering algorithm such that, among the set of filtering methods analyzed during our experimentation, is the one which gave the better performance both in CPU time and number of consistency checks. The proposed filtering algorithm works in two main steps.

1. It applies the incremental forward checking method described in Section 3.2. When at least one variable domain becames empty, *CheckConsistency* returns *false*, otherwise the following second step is executed.
2. The set of not assigned variables which are modified by the application of the first step is considered, and used to inizialize the propapagation queue Q of an *arc-consistency* filtering method. The filtering method is executed to remove further values, in the case at least one variable domain becames empty, *CheckConsistency* returns *false*, otherwise returns *true*.

Figure 2 shows the *arc-consistency* filtering algorithm. It takes as an input the set Q_{init} of modified variables and applies the 2-consistency filtering startegy by the *Revise* operator which is the core of the method. In this case the operator has the following definition: Revise(c_i, c_j) removes from the domains D_{c_i} and D_{c_j} each value $x_i - y_i \leq r_i$ which does not have *support*. That is, a value $x_i - y_i \leq r_i$ is removed from the domain D_{c_i} when there is no value $x_j - y_j \leq r_j$ in the set D_{c_j} such that $r_i + d_{x_i y_j} + r_j + d_{x_j y_i} \geq 0$ holds. When the procedure stops, it returns the set of variables with reduced domain of values.

In the experimental section we compare this strategy with the other ones, trying to find some conclusions about the relations among the number of consistency checks (we consider the test $r_i + d_{x_i y_j} + r_j + d_{x_j y_i} \geq 0$ performed inside

Arc-consistency(Q_{init})
1. $Q \leftarrow Q_{init}$
2. **while** ($Q \neq \emptyset$ **and** $\nexists D_{c_i} = \emptyset$) **do begin**
3. $c_i \leftarrow$ Pop(Q)
4. **foreach** *not assigned variable* $c_j \in C$ **do**
5. $Q \leftarrow Q \cup$ Revise(c_i, c_j)
6. **end**

Fig. 2. Arc-consistency filtering algorithm.

the *Revise* operator as equivalent to a forward checking test) the total CPU time and the number of visited search nodes.

5 Experimental Evaluation

We adopt the same evaluation procedure used in [6, 7] and use the random DTP generator defined by Stergiou. DTP instances are generated according to the parameters $\langle k, n, m, L\rangle$ (k: number of disjuncts per clause, n: number of variables, m: number of disjunction (temporal constraints); L: a positive integer such that all the constants r_i are sampled in the interval $[-L, L]$). In particular, according to [6, 7] experimental sets are generated with $k = 2$, $L = 100$ and the domain of r_i is on integers not on reals as in the general definition of DTP.

Experimental results are plotted for $n \in \{10, 12, 15, 20, 25, 30\}$, where each curve represents the number of consistency checks versus the ratio $\rho = m/n$ (in both the results of Figures 3 and 4 $\rho = m/n$ is an integer value which ranges from 2 to 14). The median number of checks over 100 random samples for different values of ρ is plotted in Figures 3(a)-3(f) where three different type of results are compared: (1) the performance of the best algorithm proposed in [6] and labeled with SK; (2) the results of the SAT-based solving methods, there are two methods: the first one labeled with $TSAT_{IS(2)}$, which corresponds to the best results claimed in the work [7], and a second one, labeled with $TSAT_{IS(3)}$, which represents some new results only published on the $TSAT$ web page (see the reference [7] for the URL); (3) the performance of our constraint-based approach, in particular the curve labeled with $CSPi$ corresponds to the best results in the paper [11], and the one labeled with $CSPineg$ represent the new results obtained with the heuristic strategy definited in Section 4. Figure 4(d) plots the percentage of problems solvable by $CSPineg$ on different n. The algorithms are implemented in Common Lisp and the reported results are obtained on a SUN UltraSparc 10 (440MHz). All the results are obtained setting a timeout of 1000 seconds of CPU time.

There are several comments on the $CSPineg$ performance: (a) all the curves have the same behavior of the previous results. It is confirmed that the harder instances are obtained for $\rho \in \{6, 7\}$ and for such values the percentage of solvable problems becomes $< 10\%$. When the number of variables n increases the hardest region narrows; (b) the median number of forward checks show that $CSPineg$ significantly improves over $CSPi$. This fact shows that the new selection variable strategy is very effective, and indirectly confirms that there could be further

space for investigating improvements of the CSP approach; (c) the $CSPineg$ compares very well with the pre-existing approaches, it outperforms the others for $n \in \{10, 12, 15, 20\}$ and it is competitive with $TSAT_{IS(3)}$ for $n \in \{25, 30\}$. However, further work will be needed to clearly outperform $TSAT_{IS(3)}$ on all n. One possible direction of research is the use of more effective filtering strategies to reduce the dimension of the search tree. However, the use of a more powerful filtering strategy has a price of an higher computational time. Hence, the real problem is find a good tradeoff among number of consistency checks, number of search nodes and CPU time.

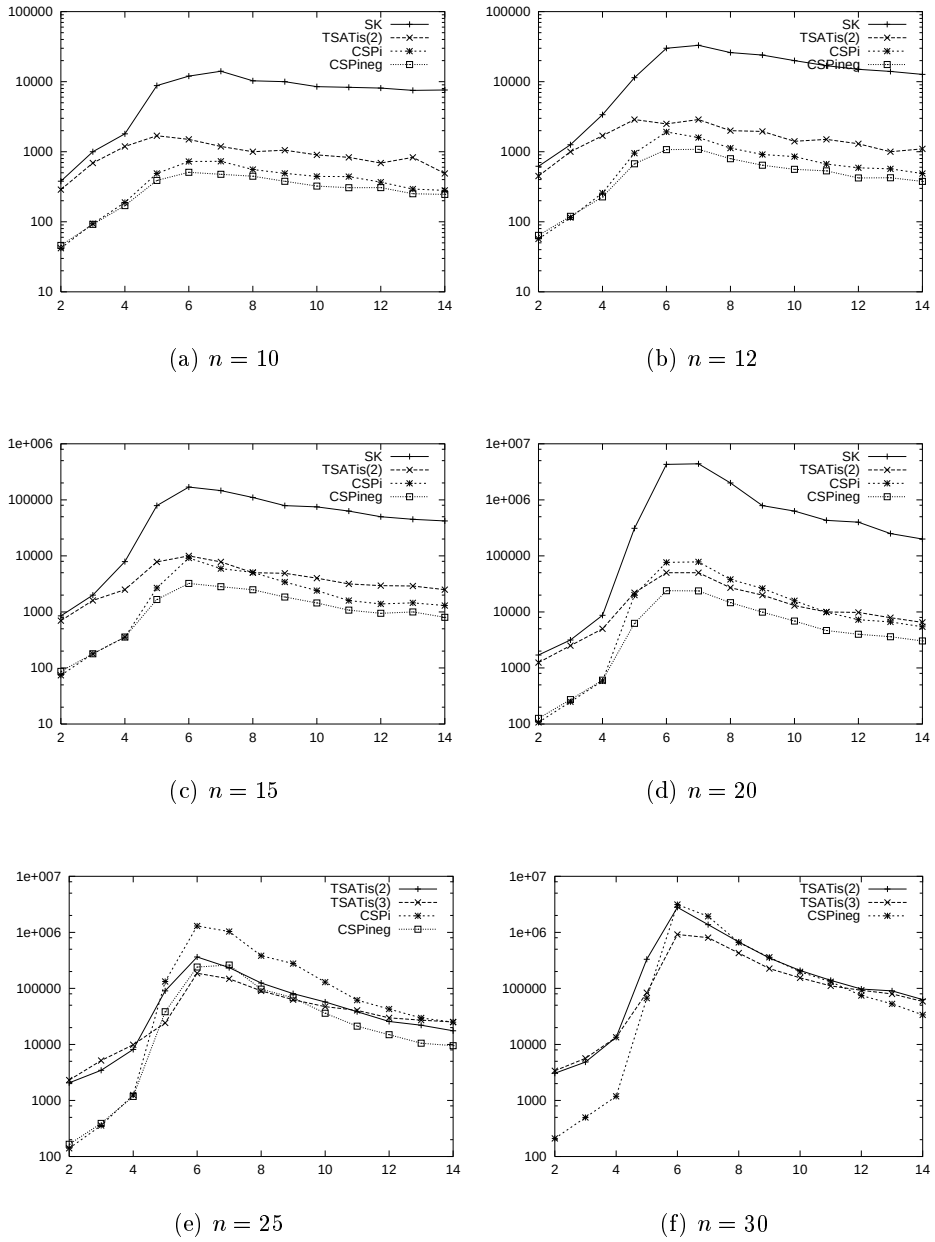
Figure 4(a) shows a comparison between the performance of our constraint-based algorithm $CSPineg$ and the other one which uses the arc-consistency filtering strategy (labeled with $CSPiac$) introduced in Section 4. With respect to number of forward checks $CSPiac$ performs about one order of magnitute worse than $CSPineg$, where in the case of the arc-consistency algorithm we consider the test $r_i + d_{x_i y_j} + r_j + d_{x_j y_i} \geq 0$ as equivalent to a forward check. On the other hand, if we consider the CPU time performance (Figure 4(c)), the ratio between the $CSPiac$ and $CSPineg$ CPU times is less than 3 in the transition phase. The analysis is completed by the results in Figure 4(d), which show that the $CSPiac$ strategy is able to reduce about 25% the number of search nodes respect to the $CSPineg$ performance.

About the results of Figure 4 we have the following observations: (a) the arc-consistency strategy performs an higher number of consistency checks respect to the forward checking strategy and many of the performed checks are unnecessary, in fact, after each solution modifications, many distances on the G_d graph remain unchanged, hence many tests of the form $r_i + d_{x_i y_j} + r_j + d_{x_j y_i} \geq 0$ are unnecessarily performed; (b) in our approch a consistency check has $O(1)$ time complexity (in the TSAT approach is at least $O(n)$) and this explain the difference in performance between number of consistency checks and CPU time shows in Figure 4.

The experimental results confirm that the CSP approch contains good ideas, in fact our results are comparable with the ones obtained by the TSAT approach which uses one of the best SAT-solver available, in addition, for lower values of the ratio $\rho = m/n$ (≤ 5) the $CSPineg$ is significantly better with respect to all the others (it is to be noted also that in many practical applications the condition $\rho \leq 5$ is likely to be verified). On the other hand, further investigation is needed to realize a competitive arc-consistency solving algorithm, in this experimentation some useful observations about tradeoffs among number of forward checks, number of search nodes explored, and CPU time are pointed out, and represent a good starting point for future research directions.

6 Conclusion

This paper has extended the constraint-based approach, initially introduced in [6] and later improved in [11], to solving the DTP temporal problem. As it is pointed out in the short discussion at the begining of the paper, DTP is going to become very relevant in many planning application. We propose two new additional solving methods for DTP. The first one is an heuristic strategy for

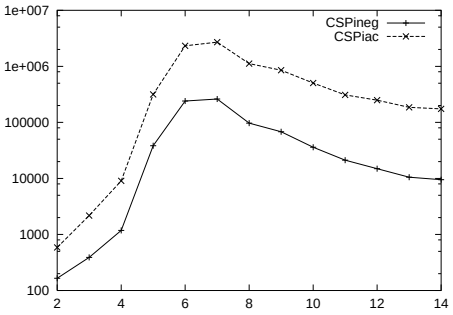


(a) $n = 10$ (b) $n = 12$

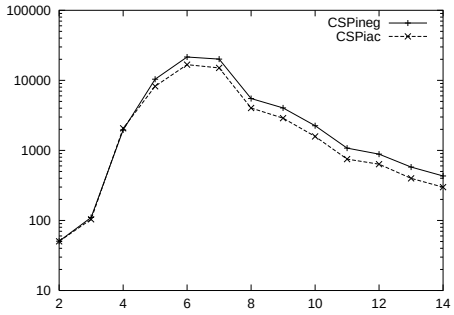
(c) $n = 15$ (d) $n = 20$

(e) $n = 25$ (f) $n = 30$

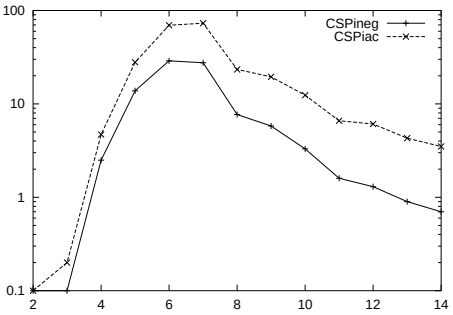
Fig. 3. Median number of forward checks for $n \in \{10, 12, 15, 20, 25, 30\}$.



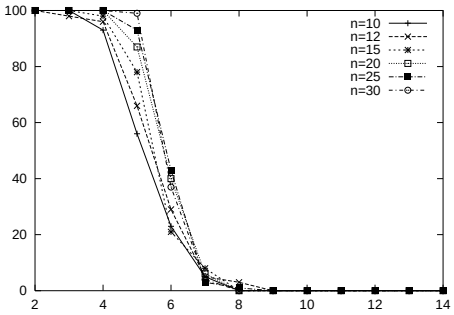
(a) Median number of forward checks for $n = 25$



(b) Average number of search nodes for $n = 25$



(c) Average CPU time in seconds for $n = 25$



(d) Percentage of solvable problems

Fig. 4. Other experimental results.

variable ordering which improves forward checking performance up to an order of magnitude respect to the results claimed in [11] and allowing a real competition with the best SAT approach. An interesting area where $CSPineg$ constantly outperforms all other approaches (when $\rho \leq 5$) emerges from an experimental evaluation. The second solving method uses a more sophisticated *arc-consistency* filtering algorithm. In this case the aim of the method is reducing the dimension of the search tree by the application of a more effective filtering strategy and to explore the possibility of finding tradeoffs among number of consistency checks, number of visited search nodes, and CPU time. The results proposed in the paper suggest that an useful research direction is the definition of an incremental version of the arc-consistency filtering algorithm.

Acknowledgments

This work is supported by ASI (Italian Space Agency) under ASI-ARS-99-96 contract and by the Italian National Research Council.

References

[1] DesJardin, M., Durfee, E., Ortiz, C., Wolverton, M.: A Survey of Research in Distributed, Continual Planning. AI Magazine **20** (1999) 13–22
[2] Pollack, M., Horty, J.: There's More to Life Than Making Plans: Plan Management in Dynamic Multiagent Environment. AI Magazine **20** (1999) 71–83
[3] Tsamardinos, I., Pollack, M.E., Horty, J.F.: Merging Plans with Quantitative Temporal Constraints, Temporally Extended Actions, and Conditional Branches. In: Proceedings of the 5th International Conference on AI Planning Systems (AIPS-2000. (2000)
[4] Stergiou, K., Koubarakis, M.: Backtracking Algorithms for Disjunctions of Temporal Constraints. Artificial Intelligence **120** (2000) 81–117
[5] Dechter, R., Meiri, I., Pearl, J.: Temporal Constraint Networks. Artificial Intelligence **49** (1991) 61–95
[6] Stergiou, K., Koubarakis, M.: Backtracking Algorithms for Disjunctions of Temporal Constraints. In: Proceedings 15th National Conference on AI (AAAI-98). (1998)
[7] Armando, A., Castellini, C., Giunchiglia, E.: SAT-based Procedures for Temporal Reasoning. In: Proceedings 5th European Conference on Planning (ECP-99). (1999) (available at `http://www.mrg.dist.unige.it/~drwho/Tsat`).
[8] Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. Computational Intelligence **9** (1993) 268–299
[9] Haralick, R., Elliott, G.: Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence **14** (1980) 263–313
[10] Cook, S., Mitchell, D.: Finding Hard Instances of the Satisfiability Problem: a Survey. In: Satisfiability Problems: Theory and Applications. DIMACS Series in Discrete Mathematics and Computer Science N.35 (1998)
[11] Oddi, A., Cesta, A.: Incremental Forward Checking for the Disjunctive Temporal Problem. In Horn, W., ed.: ECAI2000. 14th European Conference on Artificial Intelligence, IOS Press (2000) 108–111
[12] Chleq, N.: Efficient Algorithms for Networks of Quantitative Temporal Constraints. In: Proceedings of the Workshop CONSTRAINTS'95 (held in conjunction with FLAIRS-95). (1995) 40–45

September 14

Improvements to SAT-based conformant planning*

Claudio Castellini[1,2], Enrico Giunchiglia[2], and Armando Tacchella[3]

[1] Division of Informatics — Univ. of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, UK
[2] DIST — Università di Genova, Viale Causa 13, 16145, Genova, Italia
[3] Dept. of Computer Science — Rice University, 6100 Main St. MS132, 77005 Houston, Texas

Abstract. Planning as satisfiability is an efficient technique for classical planning. In previous work by the second author, this approach has been extended to conformant planning, that is, to planning domains having incomplete information about the initial state and/or the effects of actions. In this paper we present some domain independent optimizations to the basic procedure described in the previous work. A comparative experimental analysis shows that the resulting procedure is competitive with other state-of-the-art conformant planners on domains with a high degree of parallelism.

1 Introduction

Planning as satisfiability [1] is an efficient technique for classical planning. In the classical setting, the idea behind planning as satisfiability is simple: For any action description D, it is possible to compute a propositional formula tr_i^D whose satisfying assignments correspond to the possible transitions caused by the execution of an action in D. In tr_i^D there is a propositional variable A_i for each ground action symbol A, and two propositional variables F_i and F_{i+1} for each ground fluent F in D. Intuitively, F_i represents the value of F in the initial state of the transition, and F_{i+1} represents the value of F in the resulting state, after having performed the action. Then, the problem of finding a plan of length n leading from an initial state I to a goal state G corresponds to finding an assignment satisfying

$$I_0 \wedge \wedge_{i=0}^{n-1} tr_i^D \wedge G_n \tag{1}$$

(see [2] for more details). This simple idea had a lot of impact in the classical planning community, mainly because it led to very impressive results (see, e.g., [2]). In [3], the planning as satisfiability approach has been extended to conformant planning, that is, to planning problems having incomplete information about the initial state and/or the effects of actions. Some preliminary experimental results reported in [4] show that the approach can be competitive w.r.t. CGP [5], a conformant planner based on plan graphs [6].

* A special thank to Paolo Ferraris for many fruitful discussions on the topic of this paper. Paolo has also participated to the design of the architecture of $\mathcal{C}$-PLAN, and has developed the first version of $\mathcal{C}$-PLAN. Norman McCain has made possible to integrate the Causal Calculator in $\mathcal{C}$-PLAN. The first two authors are supported by ASI, CNR and MURST. The third author is partially supported by NSF grants CCR-9700061 and CCR-9988322, BSF grant 9800096, and a grant from the Intel Corporation.

In this paper we present some domain independent optimizations to the basic procedure described in [3]. Some of these optimizations have been incorporated in $\mathcal{C}$-PLAN, a SAT-based system for planning in domains whose action description component is specified using the highly expressive action language $\mathcal{C}$ [7]. As a consequence, $\mathcal{C}$-PLAN allows for conformant planning in domains with constraints, concurrent actions, and nondeterminism. A comparative experimental analysis shows that $\mathcal{C}$-PLAN is competitive with other state-of-the-art conformant planners on domains with an high degree of parallelism.

The paper is structured as follows. In Section 2 we briefly review the conformant planning via SAT approach. In Section 3 we discuss the weaknesses_of and optimizations_to the basic procedure. In Section 4 we perform the experimental comparative analysis. We end the paper in Section 5 with some conclusions.

The material here presented is part of [8]. In [8], we give the precise definitions, statements, proofs, and we also present some additional experimental analysis.

2 Conformant planning via SAT

This Section introduces some terminology and notation that will be used in the rest of the paper. Precise definitions are not given for lack of space. See [3] or [8].

We start with a set of atoms partitioned into a set of *fluent symbols* and a set of *action symbols*. A *formula* is a propositional combination of atoms. An *action* is an interpretation of the action symbols. Intuitively, to execute an action α means to execute concurrently the "elementary actions" represented by the action symbols satisfied by α. In the rest of the paper, an action α is represented by the set of action symbols satisfied by α.

An *action description* D is a finite set of expressions describing how actions change the state of the world, i.e., describing their preconditions and (possibly nondeterministic) effects. Regardless of how D is specified, we assume to have a formula tr_i^D whose satisfying assignments correspond to the transitions of D, as in the classical planning as satisfiability framework outlined in the introduction.[1]

A *planning problem* for D is characterized by two formulas I and G in the fluent signature, i.e., is a triple $\pi = \langle I, D, G\rangle$, where I and G encode the initial and goal state(s) respectively. A *plan* (of *length* $n \geq 0$) is a finite sequence $\alpha^1; \ldots; \alpha^n$ of actions.

Consider a planning problem $\pi = \langle I, D, G\rangle$. A plan $\alpha^1; \ldots; \alpha^n$ is *possible* for π if, starting from an arbitrarily chosen initial state, the consecutive execution of the actions $\alpha^1, \ldots, \alpha^n$ can lead to a goal state. According to the results in [9, 7], possible plans of length n correspond to the assignment satisfying (1), as in the classical setting. However, if we have incomplete information about the initial state and/or actions are non deterministic, then possible plans are not guaranteed to be valid. As pointed out in [9], in order for a plan $\alpha^1; \ldots; \alpha^n$ to be valid we have to check

- that the plan is "always executable" in any initial state, i.e., executable for any initial state and any possible outcome of the actions in the plan, and
- that any "possible result" of executing the plan in any initial state is a goal state.

[1] In [3], we use the term "causally explained transition": here we simply say transition.

$P := I_0 \wedge \wedge_{i=0}^{n-1} tr_i^D \wedge G_n$; $\quad V := I_0 \wedge State_0^D \wedge \neg Z_0 \wedge \wedge_{i=0}^{n-1} trt_i^D \wedge \neg(G_n \wedge \neg Z_n)$;

function $\mathcal{C}$-SAT() **return** $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$(*cnf*(P), {}).

function $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$(φ, μ)
if $\varphi = \{\}$ **then return** $\mathcal{C}$-SAT_TEST(μ);
if $\{\} \in \varphi$ **then return** *False*;
if { a unit clause $\{L\}$ occurs in φ } **then return** $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$(*assign*(L, φ),$\mu \cup \{L\}$);
L := { a literal occurring in φ };
return $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$(*assign*(L, φ),$\mu \cup \{L\}$) **or** $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$(*assign*$(\overline{L}, \varphi)$,$\mu \cup \{\overline{L}\}$).

function $\mathcal{C}$-SAT_TEST(μ)
α := { the set of literals in μ corresponding to action literals};
foreach {plan $\alpha^1; \ldots; \alpha^n$ s.t. each element in α is a conjunct in $\wedge_{i=0}^{n-1} \alpha_i^{i+1}$ }
if not SAT($\wedge_{i=0}^{n-1} \alpha_i^{i+1} \wedge V$) **then exit with** $\alpha^1; \ldots; \alpha^n$;
return *False*.

Fig. 1. $\mathcal{C}$-SAT, $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$ and $\mathcal{C}$-SAT_TEST.

As shown in [3], we can check if a plan $\alpha^1; \ldots; \alpha^n$ is valid for π by verifying whether

$$I_0 \wedge State_0^D \wedge \neg Z_0 \wedge \wedge_{i=0}^{n-1} \alpha_i^{i+1} \wedge \wedge_{i=0}^{n-1} trt_i^D \models G_n \wedge \neg Z_n. \qquad (2)$$

where

- $State_0^D$ is a formula representing the set of "possible initial states",
- Z is a newly introduced fluent symbol, and
- trt_i^D is a formula defined on the basis of tr_i^D.

Thus, we may divide the problem of finding a valid plan for π into two parts:

1. *generate* possible plans $\alpha^1; \ldots; \alpha^n$ by finding assignments satisfying the formula (1), and
2. *test* whether each generated plan $\alpha^1; \ldots; \alpha^n$ is valid by verifying whether (2) holds.

This is the idea behind the procedure $\mathcal{C}$-SAT in Figure 1. In the Figure, *cnf*(P) is a set of clauses corresponding to P, $\overline{L}$ is the literal complementary to L and, for any literal L and set of clauses φ, *assign*(L, φ) is the set of clauses obtained from φ by deleting the clauses in which L occurs as a disjunct, and eliminating $\overline{L}$ from the others. As it can be seen, $\mathcal{C}$-SAT is the Davis-Logemann-Loveland (DLL) procedure [10], except that, as soon as one assignment satisfying *cnf*(P) is found, the procedure $\mathcal{C}$-SAT_TEST is invoked. Indeed, each assignment satisfying *cnf*(P) corresponds to a set of possible plans, and $\mathcal{C}$-SAT_TEST checks whether any of these are valid. Since all the possible assignments satisfying *cnf*(P) are potentially generated and tested, $\mathcal{C}$-SAT is correct and complete for π, n: Any returned plan is valid for π (*correctness*); and, if *False* is returned, there is no valid plan of length n for π (*completeness*). However, $\mathcal{C}$-SAT only checks the existence of valid plans of length n. Indeed, even assuming that a plan is returned, we are not guaranteed of its optimality (we say that a plan of length n is *optimal* if it is valid and there is no valid plan of length $< n$). Thus, if we are looking for optimal plans, we have to consider $n = 0, 1, 2, 3, 4, \ldots$, and for each value of n, call $\mathcal{C}$-SAT. This is the idea behind the system $\mathcal{C}$-PLAN [4, 8].

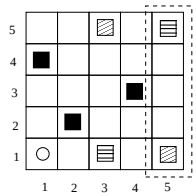


Fig. 2. A robot navigation problem

3 Improvements to SAT-Based conformant planning

In order to understand how the basic procedure above described works consider the following robot navigation problem: We are given a 5×5 grid, and 1 robot is moving in it. It starts from the bottom-left corner of the grid and its goal is to reach the right side. The robot can move $north, east, south, west$, and its duty is not trivial because there can be some objects in some locations of the grid. In order to have some nondeterminism, we assume to have one object in locations $(1, 4), (2, 2), (4, 3)$, and also that either locations $(3, 1), (5, 5)$ or $(3, 5), (5, 1)$ are occupied. Figure 2 depicts the resulting scenario. The initial position of the robot is indicated by a circle and occupied locations are marked with a black square. A dashed line outlines the goal locations. Squares filled with a pattern indicate that the corresponding locations *may* be occupied.

According to our definitions in the previous Section, we see that the shortest possible plans are of length 5: Both

$$\{east\}; \{east\}; \{east\}; \{north\}; \{east\}$$

and

$$\{east\}; \{east\}; \{north\}; \{east\}; \{east\}$$

are possible. On the other hand, any optimal valid plan, e.g.,

$$\{north\}; \{north\}; \{east\}; \{east\}; \{south\}; \{east\}; \{east\}$$

is at least 7 steps long. However, before finding a valid plan, $\mathcal{C}$-PLAN will generate and test all the possible plans of length 5, 6, and it may generate also some of the possible plans of length 7. (The possible plans of length 6 are those in which a robot does not move for one time step). Indeed, the main weaknesses of $\mathcal{C}$-PLAN are that, at each n,

1. it generates and tests *all* the possible plans. Indeed, in many cases we can generate only a subset of the possible plans, at the same time keeping completeness of the procedure, and
2. there is no interaction between generation and testing: This may lead to explore huge portions of the search space without finding a solution because of some wrong choices at the beginning of the search tree.

In the following Subsections we describe two domain independent optimizations which alleviate the first two weaknesses.

3.1 Eliminating possible plans

The basic idea is to consider only plans which are possible in a "deterministic version" of the original planning problem. In a deterministic version, all sources of nondeterminism (in the initial state and/or in the effects of the actions) are eliminated. This is possible without losing completeness, and it is a consequence of the following fact.

Consider two planning problems $\pi = \langle I, D, G\rangle$ and $\pi' = \langle I', D', G'\rangle$ in the same fluent and action signatures, and such that

1. every initial state of D' is also an initial state of D, (i.e., $I' \supset I$ is valid),
2. for every action α, the set of states of D in which α is executable is a subset of the set of states of D' in which α is executable,
3. every transition of D' is also a transition of D, (i.e., $tr^{D'} \supset tr^{D}$ is valid),
4. every goal state of π is also a goal state of π' (i.e., $G \supset G'$ is valid).

If a plan is not possible for π' then it is not valid for π. Then, in $\mathcal{C}$-PLAN we can:

- generate possible plans for π', and
- test whether each of the generated possible plans is indeed valid.

The result is still a correct and complete planning procedure (for the given planning problem π and length n). Indeed, in choosing π', we want to minimize the set of possible plans generated and then tested. Hence, we want π' to be a deterministic version of π. A planning problem $\pi' = \langle I', D', G'\rangle$ is a deterministic version of π if it also satisfies the following conditions:

1. I' is satisfied by a single state,
2. for each action α, the set of states in which α is executable in D is equal to the set of states of D' in which α is executable,
3. for any action α and state σ, there is at most one state σ' such that $\langle\sigma, \alpha, \sigma'\rangle$ is a transition of D',
4. G is equal to G'.

Of course, unless the planning problem π is already deterministic, there are many deterministic versions (possibly exponentially many). The obvious question is whether there is one which is "best" according to some criterion. Going back to our robot navigation problem, we have two deterministic versions:

- If locations $(3, 1), (5, 5)$ are occupied, the shortest plan (in the deterministic domain) has length $N_1 = 7$,
- If locations $(3, 5), (5, 1)$ are occupied, the shortest plan has length $N_2 = 5$.

Indeed, any valid plan for the original nondeterministic planning problem has length greater or equal to 7, i.e., to the max between N_1 and N_2. This is not by chance. In fact, let S be the set of deterministic versions of π. For each planning problem π' in S, let $N(\pi')$ be the length of the shortest plan for π'. Then, the length of any valid plan for π is greater or equal to

$$max_{\pi' \in S} N(\pi').$$

On the basis of this fact, we can introduce an order on S according to which $\pi' \in S$ is better than $\pi'' \in S$ if

$$N(\pi') \geq N(\pi''). \tag{3}$$

In other words, we prefer the deterministic versions which start to have solutions (each corresponding to a possible plan for the original planning problem) for the biggest possible value of n. In our robot navigation problem, this would lead us to choose the deterministic version in which the locations $(3,1),(5,5)$ are the ones which are occupied.

Determining the set S of deterministic versions of π is in general not an easy task. Even assuming that determining S is possible, finding the $\pi' \in S$ such to satisfy (3) for each $\pi'' \in S$ seems impractical at best. What we propose is the following. For simplicity, we assume to have nondeterminism only in the initial state: Analogous considerations hold for actions with nondeterministic effects. Under this assumption, we modify our $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$ procedure in Figure 1 in order to do the following:

- once an assignment μ satisfying $cnf(P)$ is found, we determine the assignment $\mu' \subseteq \mu$ to the fluent variables at time 0,
- if the possible plan corresponding to μ is not valid, and the planning problem is not already deterministic, then we disallow future assignments extending μ', by adding to I the clause consisting of the complement of the literals satisfied by μ'.

In this way, we progressively eliminate some initial states for which there is a deterministic version having a possible plan of length n. Given that some initial states have not yet been eliminated, the corresponding deterministic versions of the original planning problem have length $\geq n$. At the end, i.e., when we get to a deterministic planning problem π', π' is a deterministic version of π and it satisfies (3) for each $\pi'' \in S$.

In the example of Figure 2, this optimization has the effect of eliminating the initial state in which $\{(3,5),(5,1)\}$ are occupied. This elimination occurs immediately after the first possible but not valid plan (e.g., $\{east\};\{east\};\{east\};\{north\};\{east\}$) is found.

3.2 Incorporating Backjumping and Learning

We do not enter into the details about how backjumping and learning are implemented in SAT, and assume that the reader is familiar with the topic (see, e.g., [11–13]). Here we do the same as in [12, 13], except that we have to extend the procedure there described in order to account for the rejection of assignments corresponding to possible but not valid plans. Indeed, what we can simply do — assuming μ is an assignment corresponding to a possible but not valid plan $\boldsymbol{\alpha} = \alpha^1;\ldots;\alpha^n$ — is to return *False* and set $\vee_{i=0}^{n-1}\neg\alpha_i^{i+1}$ as the initial working reason. However, are there any other (better) choices? According to the definition of valid plan, there are two possible causes why $\boldsymbol{\alpha}$ is not valid:

1. executing $\alpha^1;\ldots;\alpha^k$ in some initial state can lead to a state σ^k, and α^{k+1} is not executable in σ^k, or
2. $\boldsymbol{\alpha}$ is always executable in any initial state, but one of the possible outcomes of executing $\boldsymbol{\alpha}$ in an initial state is not a goal state.

In both cases, $\mathcal{C}$-SAT_TEST determines an assignment μ' satisfying $\wedge_{i=0}^{n-1}\alpha_i^{i+1} \wedge V$, and thus returns *False*. Also notice that in the first case, μ' satisfies

$$\neg Z_0,\ldots,\neg Z_k, Z_{k+1},\ldots,Z_n$$

	GPT	CMBP		Cplan				
$\|P\|$-$\|T\|$	Total	#s	Total	#s	#pp	Last	Tot.search	Total
2-1	0.03	2	0.00	1	1	0.00	0.00	0.00
4-1	0.03	4	0.01	1	1	0.00	0.00	0.00
6-1	0.04	6	0.02	1	1	0.00	0.00	0.00
8-1	0.15	8	0.08	1	1	0.00	0.00	0.00
10-1	0.27	10	0.61	1	1	0.00	0.00	0.00
15-1	17.05	15	42.47	1	1	0.00	0.00	0.00
20-1	MEM	–	MEM	1	1	0.00	0.00	0.00

Table 1. Bomb in the toilet: Classic version

with $k < n$. Then we can set $\vee_{i=0}^{k} \neg\alpha_i^{i+1}$ as the initial working reason for rejecting μ: Any assignment satisfying $\wedge_{i=0}^{k} \alpha_i^{i+1}$ corresponds to a not valid plan. Of course, setting $\vee_{i=0}^{k} \neg\alpha_i^{i+1}$ as working reason for rejecting μ is better than setting $\vee_{i=0}^{n-1} \neg\alpha_i^{i+1}$: since $k < n$ each disjunct in $\vee_{i=0}^{k} \neg\alpha_i^{i+1}$ is also in $\vee_{i=0}^{n-1} \neg\alpha_i^{i+1}$, and, if $k < n-1$, the viceversa is not true.

In the example of Figure 2, this optimization has the following effect: If the possible plan $\{east\}; \{east\}; \{east\}; \{north\}; \{east\}$ is tried, then it is rejected with a reason which inhibits the further generation of possible plans beginning with $\{east\}; \{east\}$.

4 Experimental Analysis

The ideas presented in Section 2 and the optimizations described in Section 3 have been implemented in $\mathcal{C}$-PLAN. $\mathcal{C}$-PLAN accepts action descriptions specified in $\mathcal{C}$, and thus it naturally allows for, e.g., concurrency, constraints and nondeterminism. Our current version (ver. 2) of $\mathcal{C}$-PLAN has been implemented on top of SIM, an efficient SAT checker developed by our group [13].

To evaluate $\mathcal{C}$-PLAN's effectiveness we consider an elaboration of the traditional "bomb in the toilet" problem from [14]. There is a finite set P of packages and a finite set T of toilets. One of the packages is armed because it contains a bomb. Dunking a package in a toilet disarms the package and is possible only if the package has not been previously dunked. We first consider planning problems with $|P| = 2, 4, 6, 8, 10, 15, 20$, and $|T| = 1$. We compare our system with Bonet's and Geffner's GPT [15], and Cimatti's and Roveri's CMBP [16, 17] planners. These are two among the most recent conformant planners, and according to the results presented in [17], also the most effective to date. We remark that both CMBP and GPT are sequential planners: they try to execute at most one action at each step. Furthermore, CMBP computes all the valid plans, not just one like GPT and $\mathcal{C}$-PLAN. The results for these three systems are shown in Table 1. In the table, we show

- for GPT, the total time the system takes to solve the problem,
- for CMBP, the number of steps (column "#s") (i.e., the number of elementary actions) and the total time needed to solve the problem (column "Total"),
- for $\mathcal{C}$-PLAN,
  - the number of steps (i.e., the number of parallel actions) (column "#s");

- the number of possible plans generated before finding a valid one (column "#pp");
- the search time taken by the system at the last step (column "Last");
- the total search time, being the sum over all steps of the time taken by $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$ and $\mathcal{C}$-SAT_TEST (column "Tot.search"); and
- the total time taken by the system to solve the problem, excluding the off-line time necessary to compute tr_i^D and trt_i^D (column "Total"). This total time does not coincide with "Tot.search" because it includes also the times necessary for expanding the formula, and for doing some other computation internal to the system.

Times are in seconds, and all the tests have been run on a Pentium III, 850MHz, 512MBRAM running Linux SUSE 7.0. For practical reasons, we stopped the execution of a system if its running time exceeded 1200s of CPU time or if it required more than the 512MB of available RAM. In the table, the first case is indicated with "TIME" and the second with "MEM".

As it can be seen from Table 1, $\mathcal{C}$-PLAN takes full advantage of its ability to execute actions concurrently. Indeed, it solves the problem in only one step, by dunking all the packages at the same time. Furthermore, the time taken by $\mathcal{C}$-PLAN is always not measurable. CMBP and GPT have comparable performances, with CMBP being better of a factor of 2-3. However, the most interesting data about these systems is that when $|P| = 20$ they both run out of memory. Indeed, both GPT and CMBP can require huge amounts of memory: GPT for storing the set of belief states visited, and CMBP for storing (as Binary Decision Diagrams, BDD [18]) the transition relation and the set of belief states visited.

We also consider the elaboration of the "bomb in the toilet" in which dunking a package clogs the toilet. There is the additional action of flushing the toilet, which is possible only if the toilet is clogged. The results are shown in Table 2 for $|P| = 2, 4, 6, 8, 10$ and $|T| = 1, 5, 10$. With one toilet, these problems are the "sequential version" of the previous. With multiple toilets they are similar to the "BMTC" problems in [17]. As we can see from Table 2, when there is only one toilet $\mathcal{C}$-PLAN "Total" time increases rapidly compared to the other solvers. Indeed, $|T| = 1$ represents the purely sequential case in which the only valid plan consists in repeatedly dunking a package and flushing the toilet till all the packages have been dunked. On the other hand, we see, e.g., for $|P| = 6$ and $|T| = 1$, that most of $\mathcal{C}$-PLAN time is not spent in the search. By analysing these numbers and profiling $\mathcal{C}$-PLAN code, we discovered that

– most of the search time is spent by $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$: on all the experiments we tried, each call of $\mathcal{C}$-SAT_TEST takes an hardly measurable time,
– the time taken by the system to expand the formula at each step can be considerable, but (since the expansion is done once for each step) does not account for the possible big differences between "Tot.search" and "Total". This is evident if we compare row 6-1 in Table 2 with row 6-1 in Table 3: The formulas to expand in the two cases are equal, and the number of expansion is the same. However, the "Total" time in Table 2 is significantly bigger than the "Total" time in Table 3,
– the possible big differences are due to the fact that, in our current version, each time $\mathcal{C}$-SAT_TEST is invoked by $\mathcal{C}$-SAT_GEN$_{\mathrm{DLL}}$, we pay a cost linear in the size of the V formula. This cost is due to the copying of the V formula from one data-structure

	GPT	CMBP		Cplan				
$\|P\|$-$\|T\|$	Total	#s	Total	#s	#pp	Last	Tot.search	Total
2-1	0.10	3	0.00	3	6	0.00	0.00	0.01
2-5	0.04	2	0.01	1	1	0.00	0.00	0.00
2-10	0.05	2	0.03	1	1	0.00	0.00	0.00
4-1	0.04	7	0.00	7	540	0.12	0.15	0.65
4-5	0.23	4	0.79	1	1	0.00	0.00	0.00
4-10	2.23	4	11.30	1	1	0.00	0.00	0.01
6-1	0.09	11	0.04	11	52561	15.39	49.39	221.55
6-5	3.29	7	16.80	3	98346	56.92	57.34	419.53
6-10	74.15	–	MEM	1	1	0.00	0.00	0.01
8-1	0.41	15	0.20	–	–	–	–	TIME
8-5	32.07	11	112.48	–	–	–	–	TIME
8-10	MEM	–	MEM	1	1	0.00	0.00	0.01
10-1	2.67	19	1.55	–	–	–	–	TIME
10-5	MEM	15	974.45	–	–	–	–	TIME
10-10	MEM	–	MEM	1	1	0.00	0.00	0.04

Table 2. Bomb in the toilet: Multiple toilets, clogging, one bomb.

to another internal of SIM. This linear cost, multiplied by the number of times $\mathcal{C}$-SAT_TEST is invoked, accounts for the difference between "Tot.search" and "Total" in the example 6-1 of Table 2.

We remark that the potentially exponential cost of verifying a plan does not arise in practice, at least on all the experiments we tried. As a matter of fact, each time a plan is verified, the corresponding set of unit clauses is added to the V formula and (if the plan is not valid) the empty set of clauses is generated after very few splits. This was expected (see the Section on implementation in [3]): what we underestimated is the cost paid because of copying the V formula. We believe that this cost is also responsible for many of $\mathcal{C}$-PLAN's timeouts, and that a better engineering of the system, meant to solve this particular problem, will allow $\mathcal{C}$-PLAN to be even more competitive. In any case, $\mathcal{C}$-PLAN's performances are not too bad compared to the ones of the other solvers: $\mathcal{C}$-PLAN, CMBP, GPT do not solve 4, 3, 3 problems respectively.

Finally, we consider the same problem as before, except that we do not know how many packages are armed. These problems, with respect to the ones previously considered, present a higher degree of uncertainty. We consider the same values of $|P|$ and $|T|$ and report the same data as before. The results are shown in Table 3.

Contrarily to what could be expected, $\mathcal{C}$-PLAN performances are much better on the problems in Table 3 than on those in Table 2. This is most evident if we compare the number of plans generated and tested by $\mathcal{C}$-PLAN before finding a solution. For example, if we consider the four packages and one toilet problem,

- with one bomb, as in Table 2, $\mathcal{C}$-PLAN generates 540 possible plans and takes 0.65s to solve the problem (0.15s of search time),
- with possibly multiple bombs, as in Table 3, $\mathcal{C}$-PLAN generates 15 possible plans and takes 0.02s to solve the problem (0.02s is also the search time).

	GPT	CMBP		Cplan				
$\|P\|$-$\|T\|$	Total	#s	Total	#s	#pp	Last	Tot.search	Total
2-1	0.03	3	0.00	3	3	0.00	0.00	0.00
2-5	0.04	2	0.00	1	1	0.00	0.00	0.00
2-10	0.24	2	0.02	1	1	0.00	0.00	0.02
4-1	0.17	7	0.01	7	15	0.01	0.02	0.02
4-5	0.06	4	0.54	1	1	0.01	0.00	0.01
4-10	0.38	4	7.13	1	1	0.02	0.00	0.02
6-1	0.08	11	0.03	11	117	0.25	1.39	2.01
6-5	0.33	7	10.71	3	48	0.62	0.66	1.36
6-10	7.14	–	MEM	1	1	0.00	0.00	0.00
8-1	0.06	15	0.17	15	1195	12.23	147.25	184.29
8-5	2.02	11	90.57	3	2681	14.84	15.60	317.13
8-10	MEM	–	MEM	1	1	0.00	0.00	12.68
10-1	0.21	19	1.02	–	–	–	–	TIME
10-5	12.51	15	591.33	–	–	–	–	TIME
10-10	MEM	–	MEM	1	1	0.00	0.00	0.06

Table 3. Bomb in the toilet: Multiple toilets, clogging, possibly multiple bombs.

To understand why, consider for simplicity the case in which there is only one toilet and two packages P_1 and P_2. For $n = 0$

- there are no possible plans if we know that there is one bomb, and
- there is the possible plan consisting of the empty sequence of actions, corresponding to assuming that neither P_1 nor P_2 is armed, in the case we know nothing. This plan is not valid, and, because of the determinization, $\mathcal{C}$-PLAN adds a clause to the initial state saying that at least one package is armed.

For $n = 1$, $\mathcal{C}$-PLAN in both cases tries 2 possible plans. If we assume that it generates first the plan in which P_1 is dunked

- if we further assume that there is one bomb, it rejects it, and —because of the determinization— it adds a clause to the initial state which allows to conclude that the bomb is in P_2. Then, for $n = 2$ and $n = 3$, any plan in which P_2 is dunked is possible.
- in the other case, it rejects it, and —because of the determinization— it adds a clause to the initial state saying that the bomb is not in P_1 or is in P_2. Then, it generates the other plan in which only P_2 is dunked. Also this plan is rejected and a clause saying that a bomb is in P_1 or not in P_2 is added to the initial state. Thus, there is now only one initial state satisfying all the constraints: namely the one in which both P_1 and P_2 are armed. This allows $\mathcal{C}$-PLAN to conclude that there are no possible plans for $n = 2$, and to immediately generate a valid plan at $n = 3$.

The optimizations described in Section 3 help a lot. Indeed, if we consider the four packages and one toilet problem, and disable the optimizations,

- if we have one bomb, as in Table 2, $\mathcal{C}$-PLAN generates 2145 possible plans and takes 0.54s to solve the problem (0.24s in the last step),

– if we have possibly multiple bombs, as in Table 3, $\mathcal{C}$-PLAN generates 3743 possible plans and takes 0.93s to solve the problem (0.72s in the last step).

Besides $\mathcal{C}$-PLAN's performances, also CMBP and GPT seem to get benefits by the added nondeterminism. Overall, $\mathcal{C}$-PLAN, CMBP and GPT do not solve respectively 2, 3 and 2 of the considered problems. On the ones they solve, we get roughly the same picture that we had before: $\mathcal{C}$-PLAN takes full advantage of its ability to concurrently execute actions, and thus behaves better on problems with multiple toilets.

Overall, GPT, CMBP and $\mathcal{C}$-PLAN do not solve 6, 7, 6 respectively of the 37 examples that we tried.

5 Conclusions and related work

We have presented some optimizations to the basic procedure for conformant planning described in [3]. The procedure and the optimizations have been implemented in $\mathcal{C}$-PLAN ver. 2, a SAT-based conformant planner based on the SIM SAT library. $\mathcal{C}$-PLAN incorporates the Causal Calculator [9], and is thus able to reason about action descriptions specified in $\mathcal{C}$. $\mathcal{C}$ allows for, e.g., concurrency, constraints, and nondeterminism. This causes $\mathcal{C}$-PLAN to be one of most expressive conformant planners among the currently available. From the experimental analysis we get that GPT, CMBP and $\mathcal{C}$-PLAN do not solve 6, 7, 6 respectively of the 37 examples that we tried. Most important, while $\mathcal{C}$-PLAN runs out of time, CMBP and GPT run out of memory. This seems to point out that $\mathcal{C}$-PLAN range of applicability is different from the range of applicability of CMBP and GPT. Analogous results supporting this fact are reported in [19] where it is shown that for classical, highly parallel domains the planning as satisfiability approaches appear to do best. The fact that SAT-based approaches and BDD-based approaches have different range of applicability is also confirmed by

– previous work comparing BDD and DLL as SAT procedures, see [20],
– recent work in symbolic reachability in formal verification, see [21, 22].

Beside the already cited [15, 16], two other works on conformant planning are [5] and [24]. In [5], the authors propose an approach based on plan graphs. The underlying idea is to construct a planning graph for every possible deterministic version of the original planning problem. Constraints over planning graphs ensure conformance. The main weakness of the approach is that there can be exponentially many deterministic version, causing the creation of exponentially many planning graphs. In [24], Rintanen reduces the problem of conformant and conditional planning to the problem of deciding the satisfiability of a Quantified Boolean Formula (QBF). Our approach is similar to Rintanen's: The search performed by our generate and test procedure resembles the one of a QBF solver if run on formulas corresponding to conformant planning problems. On the other hand, by dealing with the original planning problem, we are able to introduce optimizations —like the ones described in Section 3— which take into account the nature of the original problem.

In [23], a new algorithm for conformant planning based on "heuristic-symbolic search", is proposed and the experimental results are impressive. A detailed analysis of the paper and the results is in our agenda.

References

1. Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. ECAI-92*, pages 359–363.
2. Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic and stochastic search. In *Proc. AAAI-96*, pages 1194–1201.
3. Enrico Giunchiglia. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *Proc. KR-2000*.
4. Paolo Ferraris and Enrico Giunchiglia. Planning as satisfiability in nondeterministic domains. In *Proc. AAAI-2000*.
5. David Smith and Daniel Weld. Conformant graphplan. In *Proc. AAAI-98*, pages 889–896.
6. Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proc. of IJCAI-95*, pages 1636–1642, 1995.
7. Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, pages 623–630.
8. Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism, 2001. Unpublished.
9. Norman McCain and Hudson Turner. Fast satisfiability planning with causal theories. In *Proc. KR-98*.
10. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
11. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
12. Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI-97*, pages 203–208.
13. Enrico Giunchiglia, Marco Maratea, Armando Tacchella, and Davide Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. of the International Joint Conference on Automated Reasoning (IJCAR'2001)*, 2001.
14. Drew McDermott. A critique of pure reason. *Computational Intelligence*, 3:151–160, 1987.
15. Blai Bonet and Hector Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. AIPS*, 2000.
16. Alessandro Cimatti and Marco Roveri. Conformant planning via model checking. In *Proc. ECP-99*.
17. Alessandro Cimatti and Marco Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.
18. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
19. Patrick Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proc. AIPS-2000*, pages 140–149.
20. T. E. Uribe and M. E. Stickel. Ordered Binary Decision Diagrams and the Davis-Putnam Procedure. In *Proc. of the 1st International Conference on Constraints in Computational Logics*, 1994.
21. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS-99*.
22. Fady Copty, Limor Fix, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. CAV-2001*.
23. A. Cimatti, E. Giunchiglia, M. Pistore, E. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: BDD-based + SAT-based symbolic model checking, 2001. Unpublished.
24. Jussi Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

Symbolic Techniques for Planning with Extended Goals in Non-Deterministic Domains

Marco Pistore, Renato Bettin, and Paolo Traverso

ITC-IRST
Via Sommarive 18, 38050 Povo, Trento, Italy
{pistore,bettin,traverso}@irst.itc.it

Abstract. Several real world applications require planners that deal with non-deterministic domains and with temporally extended goals. Recent research is addressing this planning problem. However, the ability of dealing in practice with large state spaces is still an open problem. In this paper we describe a planning algorithm for extended goals that makes use of BDD-based symbolic model checking techniques. We implement the algorithm in the MBP planner, evaluate its applicability experimentally, and compare it with existing tools and algorithms. The results show that, in spite of the difficulty of the problem, MBP deals in practice with domains of large size and with goals of a certain complexity.

1 Introduction

Research in classical planning has focused on the problem of providing algorithms that can deal with large state spaces. However, in some application domains (like robotics, control, and space applications) classical planners are difficult to apply, due to restrictive assumptions on the planning problem they are designed (and highly customized) for. Restrictive assumptions that result from practical experiences are, among others, the hypotheses about the determinism of the planning domain and the fact that goals are sets of final desired states (reachability goals). Several recent works address either the problem of planning for reachability goals in *non-deterministic domains* (see for instance [7, 15, 3, 10]), or the problem of planning for *temporally extended goals* that define conditions on the whole execution paths (see for instance [8, 1]). Very few works in planning relax both the restrictions on deterministic domains and on reachability goals, see, e.g., [11, 14]. These works show that planning for temporally extended goals in non-deterministic domains is theoretically feasible in a rather general framework. However, they leave open the problem of dealing in practice with the large state spaces that are a characteristic of most real world applications. Indeed, the combination of the two aspects of non-determinism and temporally extended goals makes the problem of plan generation significantly more difficult than considering one of the two aspects separately. From the one side, planning for temporally extended goals requires general synthesis algorithms that cannot be customized and optimized to the special case of reachability goals. From the other side, compared to planning for extended goals in deterministic domains, the planner has to take into account the fact that temporal properties must be checked on all the execution paths that result from the non-deterministic outcomes of actions. These two factors make *practical* planning for extended goals in *large* non-deterministic domains an open and challenging problem.

In this paper we address this problem. The starting point is the work presented in [14], which provides a theoretical framework for planning in non-deterministic domains. In [14], goals are formulas in the CTL temporal logic [9]. CTL provides the ability to express goals that take into account the fact that a plan may non-deterministically

result in many possible different executions and that some requirements can be enforced on all the possible executions, while others may be enforced only on some executions. In order to show that planning for CTL goals is feasible theoretically, [14] presents a planning algorithm that searches through an explicit representation of the state-space. This however limits its applicability to trivial examples. In this paper we describe in detail a novel planning algorithm based on symbolic model checking techniques, and evaluate it experimentally. The algorithm is a major departure from that presented in [14], both theoretically and practically. Theoretically, while [14] is an explicit-state depth-first forward search, the algorithm presented here is formulated directly by using symbolic model checking techniques. Starting from the goal formula, it builds an automaton that is then used to control the symbolic search on sets of states. From the practical point of view, the algorithm opens up the possibility to scale-up to large state spaces. We implement the algorithm in the MBP planner [2], and provide an extensive experimental evaluation. The experiments show that planning in such a general setting can still be done in practice in domains of significant dimensions, e.g., domains with more than 10^8 states, and with goals of a certain complexity. We compare MBP with SIMPLAN [11], a planner based on explicit-state search, and show the significant benefits of planning based on symbolic model checking w.r.t. explicit-state techniques. We also compare the algorithm for extended goals with the MBP algorithms presented in [6, 7], that have been customized to deal with reachability goals. The general algorithm for extended goals introduces a rather low overhead, in spite of the fact that it deals with a much more general problem.

The paper is structured as follows. Section 2 presents the basic definitions on planning for extended goals in non-deterministic domains. The core of the paper are Sections 3 and 4. The former presents the planning algorithm, while the latter describes its implementation in the MBP planner and shows the experimental evaluation. Finally, Section 5 draws some conclusions and discusses related work.

2 Non-Deterministic Domains, Extended Goals and Plans

In this section we recall briefly the basic definitions for planning with extended goals in non-deterministic domains. See [14] for examples and further explanations.

Definition 1. *A (non-deterministic) planning domain $\mathcal{D}$ is a tuple $(\mathcal{B}, \mathcal{Q}, \mathcal{A}, \rightarrow)$, where $\mathcal{B}$ is the finite set of (basic) propositions, $\mathcal{Q} \subseteq 2^{\mathcal{B}}$ is the set of states, $\mathcal{A}$ is the finite set of actions, and $\rightarrow \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$ is the transition relation. We write $q \xrightarrow{a} q'$ for $(q, a, q') \in \rightarrow$.*

The transition relation describes how an action leads from one state to possibly many different states. We require that relation $\rightarrow$ is total, i.e., for every $q \in \mathcal{Q}$ there is some $a \in \mathcal{A}$ and $q' \in \mathcal{Q}$ such that $q \xrightarrow{a} q'$. We denote with $\mathrm{Act}(q) \stackrel{\Delta}{=} \{a : \exists q'.\ q \xrightarrow{a} q'\}$ the set of the actions that can be performed in state q, and with $\mathrm{Exec}(q, a) \stackrel{\Delta}{=} \{q' : q \xrightarrow{a} q'\}$ the set of the states that can be reached from q performing action $a \in \mathrm{Act}(q)$.

Definition 2. *Let $\mathcal{B}$ be the set of basic propositions of a domain $\mathcal{D}$ and let $b \in \mathcal{B}$. The syntax of an* (extended) goal *g for $\mathcal{D}$ is the following:*

$$g ::= \top \mid \bot \mid b \mid \neg b \mid g \wedge g \mid g \vee g \mid \mathrm{AX}\, g \mid \mathrm{EX}\, g \mid \mathrm{A}(g\, \mathrm{U}\, g) \mid \mathrm{E}(g\, \mathrm{U}\, g) \mid \mathrm{A}(g\, \mathrm{W}\, g) \mid \mathrm{E}(g\, \mathrm{W}\, g).$$

We define the following abbreviations: $\mathrm{AF}\, g = \mathrm{A}(\top\, \mathrm{U}\, g)$, $\mathrm{EF}\, g = \mathrm{E}(\top\, \mathrm{U}\, g)$, $\mathrm{AG} = \mathrm{A}(g\, \mathrm{W}\, \bot)$, *and* $\mathrm{EF} = \mathrm{E}(g\, \mathrm{W}\, \bot)$.

Extended goals are expressed with CTL formulas. CTL allows us to express goals that take into account non-determinism. For instance, it is possible to express the different forms of reachability goals considered in [6, 7]: the goal $\mathrm{EF}\, b$ requires plans to have a chance of reaching a set of final desired states where b holds ("weak" planning), $\mathrm{AF}\, b$ requires plans that are guaranteed to achieve the goal ("strong" planning), and $\mathrm{A}(\mathrm{EF}\, b\, \mathrm{W}\, b)$ requires plans that try to achieve the goal with iterative trial-and-error strategies ("strong-cyclic" planning). We can express also different kinds of maintainability goals, e.g., $\mathrm{AG}\, g$ ("maintain g"), $\mathrm{AG}\, \neg g$ ("avoid g"), $\mathrm{EG}\, g$ ("try to maintain g"), $\mathrm{EG}\, \neg g$ ("try to avoid g"), and $\mathrm{AG}\, \mathrm{EF}\, g$ ("always maintain a possibility to reach g"). Moreover, reachability and maintainability requirements can be combined, like in the cases of $\mathrm{AF}\, \mathrm{AG}\, g$ ("reach a set of states where g can be maintained"). See [14] for a larger set of examples of extended goals that can be expressed in CTL.

In order to satisfy extended goals, we need to consider plans that are strictly more expressive than plans that simply map states of the world to actions to be executed, like universal plans [16], memory-less policies [3], and state-action tables [6, 7]. In the case of temporally extended goals, actions to be executed may also depend on the "internal state" of the executor, which can take into account, e.g., previous execution steps. More precisely, a plan can be defined in terms of an *action function* that, given a state and an *execution context* encoding the internal state of the executor, specifies the action to be executed, and in terms of a *context function* that, depending on the action outcome, specifies the next execution context.

Definition 3. *A* plan *for a domain $\mathcal{D}$ is a tuple $\pi = (C, c_0, act, ctxt)$, where C is a set of (execution) contexts, $c_0 \in C$ is the initial context, $act : \mathcal{Q} \times C \rightharpoonup \mathcal{A}$ is the action function, and $ctxt : \mathcal{Q} \times C \times \mathcal{Q} \rightharpoonup C$ is the context function.*

If we are in state q and in execution context c, then $act(q, c)$ returns the action to be executed by the plan, while $ctxt(q, c, q')$ associates to each reached state q' the new execution context. Functions *act* and *ctxt* may be partial, since some state-context pairs are never reached in the execution of the plan.

The execution of a plan results in a change in the current state and in the current context. It can therefore be described in terms of transitions between state-context pairs, like $(q, c) \xrightarrow{a} (q', c')$. Due to the non-determinism of the domain, a plan may lead to an infinite number of different executions. In [14] a finite presentation of all possible executions is obtained by defining an *execution structure*. It is a Kripke structure [9] whose set of states is the set state-context pairs, and whose transitions are the transitions of the plan. According to [14], a plan satisfies a goal g if CTL formula g holds on the execution structure corresponding to the plan.

3 The Symbolic Planning Algorithm

A planning problem requires to build a plan that satisfies the goal and is compatible with the given domain (i.e., it is executable). The algorithm we propose works by building an automaton, called the *control automaton* that is used to guide the search of a plan. The states of the control automaton are the contexts of the plan that is being built, and the transitions represent the possible evolutions of the contexts when actions are executed. Control automata are strictly related to the tree automata proposed in [12] as the basic structure for performing CTL synthesis. The outline of the symbolic planning algorithm is the following:

function *symbolic-plan*(g_0) : *Plan*
  aut := *build-aut*(g_0)
  assoc := *build-assoc*(*aut*)
  plan := *extract-plan*(*aut*, *assoc*)
  return *plan*

It works in three main steps. In the first step, *build-aut* constructs the control automaton for the given goal. In the second step, *build-assoc* exploits the control automaton to guide the symbolic exploration of the domain, and associates a set of states in the planning domain to each state in the control automaton. Intuitively, these are the states for which a plan exists from the given context. In the third step, *extract-plan* constructs a plan by exploiting the information on the states associated to the contexts.

Control automata. Control automata are the key element for the definition of the planning algorithm.

Definition 4. *A* control automaton *is a tuple* $A = (C, c_0, T, R)$*, where:*

- C *is the set of control states (or contexts), and* $c_0 \in C$ *is the initial control state.*
- $T : C \to \mathcal{P}(\mathit{Prop}(\mathcal{B}) \times \mathcal{P}(C) \times C)$ *is the transition function, where* $\mathit{Prop}(\mathcal{B})$ *is any propositional formula constructed from basic propositions* $b \in \mathcal{B}$*.*
- $R = \{B_1, \ldots, B_n\}$*, with* $B_i \subseteq C$*, is the set of the* red blocks *of the automaton.*

The transitions in $T(c)$ describe the different valid evolutions from context c. For each $(P, Es, A) \in T(c)$, component P constrains the states where the transitions is applicable, while components Es and A describe the contexts that must hold in the next states, according to the transition. More precisely, each context $E \in Es$ must hold for "some" of the next states, while A defines the context that must hold for "all the other" next states. The distinction between contexts Es and context A is necessary in the case of non-deterministic domain, since it permits to distinguish between behaviors that the plan should enforce on all next states, or only on some of them.

Component R defines conditions on the valid infinite executions of a plan. These coincide with the so called "acceptance conditions" of automata theory [13]. Acceptance conditions are necessary to distinguish the control states of a plan where the execution can persist forever from the control states that should be left eventually in order to allow for a progress in the fulfillment of the goal. More precisely, a given red block $B \in R$ is used to represent all the control states in which the execution is trying to reach or achieve a given condition. If an execution of a plan persists inside B, then the condition is never reached, the execution is not accepted and the plan is not valid. If a control state does not appear in any red block, then it corresponds to a situation where only safety or maintainability goals have to be fulfilled, so no progress is required.

Construction of the control automata. We now define how the control automaton is constructed from the goal.

Definition 5. *The control automaton* $A =$ *build-aut*(g_0) *is built according to rules:*

- $c_0 = [g_0] \in C$.
- *If* $[g_1, \ldots, g_n] \in C$ *then, for each* $(P, EX, AX) \in$ *progr*$(g_1 \wedge \cdots \wedge g_n)$ *and for each partition* $\{EX_1, \ldots, EX_n\}$ *of* EX*:*

  $$(P \ , \ \{\textit{order-goals}(AX \cup EX_i) : i = 1..n\} \ , \ \textit{order-goals}(AX) \ ) \ \in \ T(c).$$

  Moreover, order-goals$(AX \cup EX_i) \in C$ *for* $i = 1..n$ *and* order-goals$(AX) \in C$*.*
- *For each strong until subgoal* g *of* g_0*, let* $B_g = \{c \in C : \mathbf{head}(c) = g\}$*; if* $B_g \neq \emptyset$*, then* $B_g \in R$*.*

Each state of the control automaton corresponds to an ordered list of subgoals, representing those subgoals that the executor should currently achieve. The order of the subgoals represents their priorities. Indeed, there are situations in which it is necessary to distinguish control states that correspond to the same subgoals according to their priorities. Consider for instance the case of goal $g_{\mathrm{AG}} = \mathrm{AG}\,(\mathrm{AF}\,p \wedge \mathrm{AF}\,q)$, that requires to keep achieving both condition p and condition q. Two different contexts generated in the construction of the control automaton for this goal are $[\mathrm{AF}\,p, \mathrm{AF}\,q, g_{\mathrm{AG}}]$ and $[\mathrm{AF}\,q, \mathrm{AF}\,p, g_{\mathrm{AG}}]$. They have the same subgoals, but the first one gives priority to goal $\mathrm{AF}\,p$, while the second one gives priority to goal $\mathrm{AF}\,q$. By switching between the two contexts, the plan guarantees that both the conditions p and q are achieved infinitely often. In general, the first goal **head**(c) in a context c is the goal with the highest priority, and it is the goal that the planning algorithm is trying to achieve first.

In order to be able to define the priority of the subgoals we need to distinguish three categories of formulas: the *strong until* goals ($\mathrm{A}(\_\,\mathrm{U}\,\_)$ and $\mathrm{E}(\_\,\mathrm{U}\,\_)$), the *weak until* goals ($\mathrm{A}(\_\,\mathrm{W}\,\_)$ and $\mathrm{E}(\_\,\mathrm{W}\,\_)$), and the *transient* goals ($\mathrm{AX}\,\_$, $\mathrm{EX}\,\_$, $\_ \vee \_$, and $\_ \wedge \_$). A transient goal is "resolved" in one step, independently from their priority: prefixes AX and EX, for instance, express conditions only on the next execution step. Weak until goals are allowed to hold forever, without being resolved. Therefore, we assign a low priority to transient and weak until goals. Strong until goals must be instead eventually resolved for the plan to be valid. We assign a high priority to these goals, and, among the strong until goals, we give priority to the goals that are active since more steps. Namely, the longer a strong until goal stays active and unresolved, the "more urgent" the goal becomes. In Definition 5 the ordering of subgoals sg is performed by function *order-goals*(sg, c). The input context c represents the list of the subgoals in the old context; it is necessary to determine the priority among the strong until goals that are already active in the old context.

One of the key steps in the construction of the transition function of a control automaton is the function *progr*. It associates to each goal g the conditions that g defines on the current state and on the next states to be reached, according to the CTL semantics. This is obtained by unrolling the weak and strong until operators, as shown by the following rules:

- *progr*$(\mathrm{A}(g\,\mathrm{U}\,g')) = (\textit{progr}(g) \wedge \mathrm{AX}\,\mathrm{A}(g\,\mathrm{U}\,g')) \vee \textit{progr}(g')$ and
  progr$(\mathrm{E}(g\,\mathrm{U}\,g')) = (\textit{progr}(g) \wedge \mathrm{EX}\,\mathrm{E}(g\,\mathrm{U}\,g')) \vee \textit{progr}(g')$;
- *progr*$(\mathrm{A}(g\,\mathrm{W}\,g')) = (\textit{progr}(g) \wedge \mathrm{AX}\,\mathrm{A}(g\,\mathrm{W}\,g')) \vee \textit{progr}(g')$ and
  progr$(\mathrm{E}(g\,\mathrm{W}\,g')) = (\textit{progr}(g) \wedge \mathrm{EX}\,\mathrm{E}(g\,\mathrm{W}\,g')) \vee \textit{progr}(g')$.

Function *progr* commutes with the other operators: e.g., *progr*$(g \wedge g') =$ *progr*$(g) \wedge$ *progr*(g'). For instance, *progr*$(\mathrm{AG}\,p) = p \wedge \mathrm{AX}\,\mathrm{AG}\,p$, *progr*$(\mathrm{EF}\,q) = q \vee \mathrm{EX}\,\mathrm{EF}\,p$, and *progr*$(\mathrm{AG}\,p \wedge \mathrm{EF}\,q) = (p \wedge q \wedge \mathrm{AX}\,\mathrm{AG}\,p) \vee (p \wedge \mathrm{AX}\,\mathrm{AG}\,p \wedge \mathrm{EX}\,\mathrm{EF}\,q)$. We can assume that formula *progr*(g) is written in disjunctive normal form, as in the examples above. Each disjunct corresponds to an alternative evolution of the domain, i.e., to alternative plans we can search for. Each disjunct consists of the conjunction of three kinds of formulas, the propositional ones b and $\neg b$, those of the form $\mathrm{EX}\,f$, and those of the form $\mathrm{AX}\,h$. In the algorithm, we make this structure explicit and represent *progr*(g) as a set of triples

$$\textit{progr}(g) = \{(P_i, EX_i, AX_i) \mid i \in I\}.$$

where $p \in P_i$ are the propositional formulas of the i-th disjunct of *progr*(g), and $f \in EX_i$ ($h \in AX_i$) if $\mathrm{EX}\,f$ ($\mathrm{AX}\,h$) belongs to the i-th disjunct.

In the case the component EX of a disjunct (P, EX, AX) contains more than one subgoal, the generation of the control automaton has to take into account that there are different ways to distribute the subgoals in EX to the set of next states. For instance, if set EX contains two subgoals, then we can require that both the subgoals hold in the same next state, or that they hold in two distinct next states. In the general case, any partition $EX_1, \ldots, EX_n$ of the subgoals in EX corresponds to a possible way to associate the goals to the next states. Namely, for each $i = 1..n$, there must be some next state where subgoals $AX \cup EX_i$ hold. In all the other states, subgoals in AX must hold.

A plan for a given goal is not valid if it allows for executions where a strong until goals becomes eventually active and is then never resolved. In order to represent these undesired behaviors, the construction of the automaton generates a red block B_g for each set of contexts that share the same "higher-priority" strong until goal g.

Associating states to contexts. Once the control automaton for a goal g_0 is built, the planning algorithm proceeds by associating to each context in the automaton a set of states in the planning domain. The association is built by function *build-assoc*:

```
1 function build-assoc(aut) : Assoc
2   foreach c ∈ aut.C do assoc[c] := Q
3   green-block := {c ∈ C : ∀B ∈ aut.R . c ∉ B}
4   blocks := aut.R ∪ {green-block}
5   while (∃B ∈ blocks . need-refinement(B)) do
6     if B ∈ aut.R then foreach c ∈ B do assoc[c] := ∅
7     while (∃c ∈ B . need-update(c)) do
8       assoc[c] := update-ctxt(aut, assoc, c)
9   return assoc
```

The algorithm starts with an optimistic association, that assigns all the states Q in the domain to each context (line 2). The association is then iteratively refined. At every iteration of the loop (lines 5-8), a block of contexts is chosen, and the corresponding associations are updated. Those states are removed from the association, from which the algorithm discovers that the goals in the context are not satisfiable. The algorithm terminates when a fixpoint is reached, that is, whenever no further refinement of the association is possible: in this case, function *need-refinement*(B) at line 5 evaluates to false for each $B \in$ *blocks* and the guard of the **while** fails. The chosen block of contexts may be either one of the red blocks, or the block of states that are not in any red block (this is the "green" block of the automaton). In the case of the green block, the refinement step must guarantee only that all the states associated to the contexts are "safe": that is, they never lead to contexts where the goal cannot be achieved anymore. This refinement (lines 7-8) is obtained by choosing a context in the green block and by "refreshing" the corresponding set of states (function *update-ctxt*). Once the fixpoint is reached and all the refresh steps on the states in B do not change the association (i.e., no context in B needs updates), the loop at lines 7-8 is left, and another block is chosen. In the case of a red block, not only does the refinement guarantee that the states in the association are "safe", but also that the contexts in the red block are eventually left. Indeed, as we have seen, executions that persist forever in the control states of a red block are not valid. To this purpose, the sets of states associated to the red-block are initially emptied (line 6). Then, iteratively, one of the control states in the red-block is chosen, and its association is updated (lines 7-8). In this way, a least fixpoint is computed for the states associated to the red block.

The core step of *build-assoc* is function *update-ctxt*$(aut, assoc, c)$. It takes as input the automaton, $aut = (C, c_0, T, R)$, the current association of states *assoc* and a context $c \in C$, and returns the new set of states to be associated to c.

$$\begin{aligned} \textit{update-ctxt}(aut, assoc, c) \triangleq \{ & q \in \mathcal{Q} : \\ & \exists a \in \mathcal{A},\ \exists (P, Es, A) \in T(c) \\ & q \in \textit{states-of}(P) \wedge \\ & (q, a) \in \textit{strong-pre-image}(assoc[A]) \wedge \\ & (q, a) \in \textit{multi-weak-pre-image}(\{assoc[E] : E \in Es\}) \}. \end{aligned}$$

For a state to be associated to the context, the next states corresponding to the execution of some action $a \in \mathcal{A}$ should satisfy the transition conditions of the automaton. Let us consider an action a and an element $(P, Es, A) \in T(c)$. Formula P describes conditions on the current states. Only those states that satisfy property P are valid (condition $q \in \textit{states-of}(P)$). A is the context that should be reached for "all the other" next states, i.e., for all the next states not associated with any context in Es. Since all the contexts in Es contain a superset of the goals in context A, we check, without loss of generality, that *all* the next states are valid for context A. In order to satisfy this constraint, function *strong-pre-image* is exploited on the set $assoc[A]$ of states that are associated to context A. Function *strong-pre-image*(Q) returns the state-action pairs that guarantee to reach states in Q:

$$\textit{strong-pre-image}(Q) \triangleq \{(q, a) : a \in \mathrm{Act}(q) \wedge \mathrm{Exec}(q, a) \subseteq Q\}.$$

Set Es contains the contexts that must be reached for some next states. To satisfy this constraint, function *multi-weak-pre-image* is called on the set $\{assoc[E] : E \in Es\}$ whose elements are the sets of states that are associated to the contexts in Es. Function *multi-weak-pre-image* returns the state-action pairs that guarantee to cover all the sets of states received in input:

$$\begin{aligned} &\textit{multi-weak-pre-image}(Qs) \triangleq \\ &\quad \{(q, a) : a \in \mathrm{Act}(q) \wedge \exists i : Qs \mapsto \mathrm{Exec}(q, a)\ .\ \forall Q \in Qs\ .\ i(Q) \in Q\}. \end{aligned}$$

This function can be seen as a generalization of function *weak-pre-image*(Q), that computes the state-action pairs that may lead to a state in Q: *weak-pre-image*$(Q) = \{(q, a) : \mathrm{Exec}(q, a) \cap Q \neq \emptyset\}$. Indeed, in function *multi-weak-pre-image*(Qs) an injective map is required to exist from the Qs to the next states obtained by the execution of the state-action pair. This map guarantees that there is at least one next state in each set of states is Qs. We remark that function *update-ctxt* is the critical step of the algorithm, in terms of performance. Indeed, this is the step where the domain is explored to compute pre-images of sets of states. BDD-based symbolic techniques [4] are exploited in this step to obtain a compact representation of the sets of states associated to the contexts, and to allow for an efficient exploration of the domain.

Extracting the plan Once the association *assoc* from contexts to sets of states is built for automaton *aut*, a plan can be easily obtained. The set of contexts for the plan coincides with the set of contexts of the control automaton *aut*. The information necessary to define functions *act* and *ctxt* is implicitly computed during the execution of the function *build-assoc*. Indeed, functions *update-ctxt* and *multi-weak-pre-image* determine, respectively, the action *act*(q, c) to be performed from a given state q in a given context c, and the next execution context *ctxt*(q, c, q') for any possible next state q'. A plan can thus be obtained from a given assignment by executing one more step of the refinement function and by collecting these information.

4 Experimental Evaluation

We have implemented the planning algorithm inside the MBP planner. MBP [2] is built on top of a state-of-the-art symbolic model checker, NuSMV [5]. Further information on MBP can be found at `http://sra.itc.it/tools/mbp/`.

The experimental evaluation is designed to test the scalability of the approach, both in terms of the size of the domain and in terms of the complexity of the goal. In the experiments we also draw a comparison with planning algorithms for extended goals based on an explicit-state exploration of the domain, and in particular with SimPlan. SimPlan [11] implements different approaches to planning in non-deterministic domains. We focus on the logic-based planning component, where extended goals can be expressed in (an extension of) Linear Temporal Logic (LTL). LTL formulas can be used in SimPlan to describe user-defined, domain- and goal-dependent control strategies, that can provide aggressive pruning of the search space. In the experiments we test the performance of SimPlan with and without strategies. Another important comparison term are the algorithms provided by MBP for the specific case of reachability goals [6, 7]. Some of the experiments are designed to evaluate the overhead of the general algorithm for extended goals w.r.t. the optimized algorithms.

We consider the "robot delivery" planning domain, first described in [11]. This domain describes a building, composed by 8 rooms connected by 7 doors. A robot can move from room to room, picking up and putting down objects. Some rooms in the domain may be designed as "producer" and as "consumer" rooms: an object of a certain type can disappear if positioned in the corresponding consumer room, and can then reappear in one of the producer rooms. Furthermore, in order to add non-determinism to the system, some of the doors may be designed to close without intervention of the robot: they are called "kid-doors" in [11].

The experiments have been performed on a Pentium III 700 MHz with 4 Gb RAM of memory running Linux. The time limit was set to 1 hour (3600 seconds). All the experiments have been run on 5 random instances. In the tables, we report the average time required to complete. In the case of MBP, the reported times include also the preprocessing time necessary in MBP to build the symbolic representation of the planning domain. In the case only some of the instances terminate in the time limit, we report the average on the instances that terminate and the number t of terminated instances as $[t/5]$. If all the instances of an experiment do not terminate, the corresponding cell is left empty.

The first two experiments coincide with the experiments proposed in [11]. **Experiment 1** consists in moving objects into given rooms and then maintain them there. No producer and consumer rooms are present in this experiment. We fix the number of objects present in the domain to 5. The number n of objects to be moved ranges from 1 to 5, while the number k of kid-doors ranges from 0 to 7. The CTL goal is the following:

$$\mathrm{AF\,AG}\,(\mathit{in}(\mathit{obj}_1, \mathit{room}_1) \wedge \cdots \wedge \mathit{in}(\mathit{obj}_n, \mathit{room}_n)).$$

Experiment 2 consists in reactively delivering produced objects to the corresponding consumer room. The number p of producer and consumer rooms ranges 1 to 4, while the number k of kid-doors ranges form 0 to 7. The CTL goal is the following:

$$\mathrm{AG}\,\big(\bigwedge\nolimits_{i=1..p} \mathit{in}(\mathit{obj}_i, \mathit{prod}_i) \rightarrow \mathrm{AF}\,(\mathit{in}(\mathit{obj}_i, \mathit{cons}_i))\big)$$

The results of these two experiments are reported in Tables 1 and 2 for MBP, for SimPlan with control strategies (SimPlan with CS), and for SimPlan without control formulas (SimPlan w/o CS). MBP and SimPlan exhibit complementary behaviors

	MBP					SIMPLAN with CS					SIMPLAN w/o CS		
	n = 1	n = 2	n = 3	n = 4	n = 5	n = 1	n = 2	n = 3	n = 4	n = 5	n = 1	n = 2	n = 3
k = 0	0.7	3.4	22.1	143.9	1094.6	0.5	0.7	1.2	1.6	1.7	311.9	1145.8 [1/5]	-
k = 1	0.7	4.5	33.7	195.3	1219.6	1.0	1.9	6.3	4.8	9.1	106.5 [2/5]	0.5 [1/5]	-
k = 2	0.8	5.0	38.9	275.1	1648.2	8.4	11.4	11.5	116.7	128.6 [4/5]	-	-	-
k = 3	0.8	6.4	41.2	276.9	2163.2 [2/5]	16.0	40.1 [4/5]	378.9 [3/5]	727.0 [3/5]	-	-	-	-
k = 4	1.0	5.6	45.7	336.9	2185.3 [3/5]	22.2	1478.5 [3/5]	275.7 [2/5]	-	-	-	-	-
k = 5	1.2	7.8	43.4	350.2	1866.1 [3/5]	680.5	352.8 [2/5]	420.1 [1/5]	-	-	-	-	-
k = 6	1.2	8.8	52.1	426.2	2505.1 [3/5]	1143.2 [4/5]	-	-	-	-	-	-	-
k = 7	1.4	9.4	42.7	303.3	2886.1 [2/5]	-	-	-	-	-	-	-	-

Table 1. Results of Experiment 1.

	MBP				SIMPLAN with CS				SIMPLAN w/o CS			
	p = 1	p = 2	p = 3	p = 4	p = 1	p = 2	p = 3	p = 4	p = 1	p = 2	p = 3	p = 4
k = 0	0.0	6.4	124.6	2053.6	0.3	2.4	28.7	303.6	33.2 [4/5]	721.1 [1/5]	-	-
k = 1	0.0	6.1	137.6	2426.9	0.8	15.1	360.2	309.8 [3/5]	9.2 [2/5]	17.8 [1/5]	-	-
k = 2	0.0	6.3	112.5	2684.1	15.0	63.0 [4/5]	918.0 [1/5]	-	-	-	-	-
k = 3	0.0	5.6	123.1	2063.1	245.7	3289.8 [1/5]	-	-	-	-	-	-
k = 4	0.1	12.8	130.6	2325.5	12.2 [3/5]	-	-	-	-	-	-	-
k = 5	0.1	11.3	140.6	2944.9	1386.9 [1/5]	-	-	-	-	-	-	-
k = 6	0.1	7.9	130.3	2940.2 [3/5]	1104.6 [1/5]	-	-	-	-	-	-	-
k = 7	0.1	12.4	140.8	2703.2	1.8 [1/5]	-	-	-	-	-	-	-

Table 2. Results of Experiment 2.

in these tests. The performance of MBP is left rather unaffected when kid-doors are added, but the time required to find a plan grows exponentially in the number n of objects to be moved and in the number p of producer rooms. SIMPLAN with control strategies scales linearly with respect to the number n of objects, and behaves better than MBP also when the number p of producer rooms grows. SIMPLAN, however, suffers remarkably when kid-doors are added to the domain.

Some remarks are in order on the different behaviors of the two systems in the first experiment in the case the number n of objects to be moved grows. The number of steps that the robot must perform grows linearly in the number of objects, while the number of interesting states of the planning domain grows exponentially. MBP searches for a plan for all the states in the domain. This explains its exponential grow. The search control strategies in SIMPLAN, instead, prune most of the search space, at least in the case no kid-doors are present. The plan is therefore built in linear time w.r.t. its length. When the search control strategies are disallowed in SIMPLAN, a larger portion of the state space should be explored, and the performance becomes much worse. Indeed, plans are found in the time limit only for very small values of parameters k, n, and p.

Experiment 3 is designed to compare the performance of the general extended-goals planning algorithm of MBP with the optimized algorithms provided by MBP for reachability goals. In this case, the robot is required to reach a state where a goal condition is satisfied. The goal conditions have the following form:

$$p = in(obj_1, room_1) \wedge \cdots \wedge in(obj_n, room_n).$$

	k = 0					k = 7				
	n = 1	n = 2	n = 3	n = 4	n = 5	n = 1	n = 2	n = 3	n = 4	n = 5
$\mathrm{AF}\, p$	0.7	3.8	26.9	160.2	1316.3	0.3	0.3	0.4	0.4	0.4
Strong	0.4	2.3	16.1	139.5	766.8	0.3	0.3	0.3	0.4	0.4
SIMPLAN with CS	0.4	0.7	1.2	1.6	2.1	-	-	-	-	-
SIMPLAN w/o CS	-	-	-	-	-	-	-	-	-	-
$\mathrm{A}(\mathrm{EF}\, p\, \mathrm{W}\, p)$	1.3	9.9	46.1	601.6	2547.3	2.3	12.9	75.0	589.8	-
SC-Global	0.6	3.6	26.0	266.6	1253.3	1.1	5.0	39.9	238.6	1880.6
SC-Local	0.2	1.3	9.3	111.1	615.1	2.1	15.3	152.8	1525.8	-

Table 3. Results of Experiment 3.

We consider three optimized algorithms for reachability goals. The first algorithm tries to build *Strong* plans, i.e., plans that reach condition p despite non-determinism. It corresponds to temporally extended goal $\mathrm{AF}\, p$. The other algorithms try to build *Strong-Cyclic* plans by exploiting two different approaches, the *Global* and the *Local* approach described in [6]. We recall from Section 2 that a strong-cyclic plan defines a trial-and-error strategy, corresponding to the temporally extended goal $\mathrm{A}(\mathrm{EF}\, p\, \mathrm{W}\, p)$. Strong-cyclic plans cannot be expressed in SIMPLAN: indeed, SIMPLAN is not able to express those goals that require a combination of universal and existential path quantifiers.

Strong-cyclic plans are interesting in the cases where strong plans do not exist due to the non-determinism in the domain. In order to allow for such situations, in this experiment we use a variant of the robot delivery domain, where the robot may fail to open a kid-door. (In the original domain, the robot always succeeds in opening a door; kid can close it again only from the next round.) If a kid-door is on the route of the robot, no strong plan exists for that problem: the robot can only try to open the door until it succeeds.

The results of this experiment are shown in Table 3. We report only the case of 0 kid-doors (where strong plans always exist), and the case of 7 kid-doors (where strong plans never exist). In all the cases, the number n of objects to be moved ranges from 1 to 5. The upper part of the table covers the "strong" reachability planning problem and compares the optimized MBP algorithm (Strong), the general algorithm on goal $\mathrm{AF}\, p$, and, for completeness, SIMPLAN. In the case $k = 7$, the times in Table 3 are those required by MBP to report that no strong plan exists. The lower part of the table considers the "strong-cyclic" reachability planning problem and compares the two strong-cyclic algorithms of MBP (SC-Global and SC-Local) and the general MBP algorithm on goal $\mathrm{A}(\mathrm{EF}\, p\, \mathrm{W}\, p)$. The experiment shows that the generic algorithm for temporally extended goals compares quite well with respect to the optimized algorithms provided by MBP. Indeed, the generic algorithm requires about twice the time needed by the optimized algorithm in the case of the strong plans and about $2.5\times$ the time of the optimized "global" algorithms for the strong-cyclic planning. The "local" algorithm behaves better than the generic algorithm (and than the "global" one) in the case a strong plan exists, i.e., $k = 0$; it behaves worse in the case no plan exists, i.e., $k = 7$. This difference in the performances of the MBP algorithms derives from the overhead introduced in the generic algorithm by the need of managing generic goals, and from the optimizations present in the specific algorithms. For instance, the Strong and SC-Local algorithms stop when a plan is found for all the initial states, while the generic algorithm stops only when a fixpoint is reached.

Experiment 4 tests the scalability of the algorithm w.r.t. the complexity of the goal. In particular, we consider the case of sequential reachability goals in the modified domain of Experiment 3. Given a sequence $p_1, \ldots, p_t$ of conditions to be reached, with

$$p_i = \mathit{in}(\mathit{obj}_{i,1}, \mathit{room}_{i,1}) \wedge \cdots \wedge \mathit{in}(\mathit{obj}_{i,n}, \mathit{room}_{i,n}),$$

	k = 0						k = 7					
	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
AF _	35.1	97.5	158.7	196.1	197.9	290.4	0.3	0.5	0.4	0.6	0.6	0.8
SIMPLAN with CS	1.2	2.9	4.2	6.8	7.8	11.5	-	-	-	-	-	-
SIMPLAN w/o CS	-	-	-	-	-	-	-	-	-	-	-	-
A(EF _ W _)	128.1	195.1	307.7	358.0	583.0	632.4	151.3	279.8	342.8	544.0	642.9	799.4

Table 4. Results of Experiment 4.

we consider the "strong" sequential reachability planning problems

$$\mathrm{AF}\,(p_1 \wedge \mathrm{AF}\,(p_2 \wedge \cdots \mathrm{AF}\,(p_t)))$$

and the "strong-cyclic" sequential reachability planning problem

$$\mathrm{A}(\mathrm{EF}\,(p_1 \wedge \cdots \mathrm{A}(\mathrm{EF}\,p_t\,\mathrm{W}\,p_t))\,\mathrm{W}(p_1 \wedge \cdots \mathrm{A}(\mathrm{EF}\,p_t\,\mathrm{W}\,p_t))).$$

In Table 4 we present the outcomes of the experiment. In the strong case, we present the results also for SIMPLAN. In the strong-cyclic case a comparison is not possible, as SIMPLAN is not able to represent this kind of goals. The number n of objects is set to 3, while the nesting level t ranges from 1 to 6. The cases of 0 and of 7 kid-doors are considered. The experiment shows that MBP scales about linearly in the number of nested temporal operators, both in the case of strong and in the case of strong-cyclic multiple reachability. In the case of 0 kid-doors and strong reachability, also SIMPLAN with search control strategies scales linearly, and the performance is much better than MBP. Without search strategies, SIMPLAN is not able to complete any of the tests in this experiment.

The experimental evaluation shows that MBP is able to deal with relatively large domains (some of the instances of the considered experiments have more than 10^8 states) and with high non-determinism, and that the performance scales well with respect to the goal complexity. In terms of expressiveness, CTL turns out to be an interesting language for temporally extended goals. With respect to LTL (used by SIMPLAN), CTL can express goals that combine universal and existential path quantifiers: this is the case, for instance, of the strong-cyclic reachability goals. On specific planning problems, (e.g., reachability problems) the overhead w.r.t. optimized state-of-the-art algorithms is acceptable. The comparison with SIMPLAN shows that the algorithms based on symbolic techniques outperform the explicit planning algorithms in the case search control strategies are not allowed. With search control strategies, SIMPLAN performs better than MBP in the case of domains with a low non-determinism. The possibility of enhancing the performance of MBP with search strategies is an interesting topic for future research.

5 Conclusions and Related Work

In this paper, we have described a planning algorithm based on symbolic model checking techniques, which is able to deal with non-deterministic domains and goals as CTL temporal formulas. This work extends the theoretical results presented in [14] by developing a symbolic algorithm that fully exploits the potentiality of BDDs for the efficient exploration of huge state spaces. We implement the algorithm in MBP, and perform a set of experimental evaluations to show that the approach is practical. We test the scalability of the planner depending on the dimension of the domain, on the degree of non-determinism, and on the length of the goal. The experimental evaluation gives positive results, even in the case MBP is compared with hand-tailored domain and goal dependent heuristics (like those of SIMPLAN), or with algorithms optimized to deal with reachability goals.

Besides SIMPLAN, very few attempts have been made to build planners that work in practice in such a general setting like the one we propose. The issue of "temporally extended goals" is certainly not new. However, most of the works in this direction restrict to deterministic domains, see for instance [8, 1]. Most of the planners able to deal with non-deterministic domains, do not deal with temporally extended goals [7, 15, 3].

Planning for temporally extended goals is strongly related to the "synthesis problem" [12]. Indeed, the planner has to synthesize a plan from the given goal specification. We are currently investigating the applicability of the proposed algorithm to synthesis problems.

In this paper we focus on the case of full observability. An extension of the work to the case of planning for extended goals under partial observability is one of the main objectives for future research.

References

1. F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.
2. P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a Model Based Planner. In *Proc. of IJCAI'01 workshop on Planning under Uncertainty and Incomplete Information*, 2001.
3. B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. AIPS 2000*, 2000.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.
5. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a reimplementation of SMV. In *Proc. STTT'98*, pages 25–31, 1998.
6. A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. Technical report, IRST, Trento, Italy, 2001.
7. A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proc. AAAI'98*, 1998.
8. G. de Giacomo and M.Y. Vardi. Automata-theoretic approach to planning with temporally extended goals. In *Proc. ECP'99*, 1999.
9. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 16, pages 995–1072. Elsevier, 1990.
10. R. Jensen and M. Veloso. OBBD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intellegence Research*, 13:189–226, 2000.
11. F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.
12. O. Kupferman and M. Vardi. Synthesis with incomplete informatio. In *Proc. Int. Conf. on Temporal Logic*, 1997.
13. O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Proc. CAV'94*, 1994.
14. M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-deterministic Domains. In *Proc. IJCAI'01*. AAAI Press, August 2001.
15. J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellegence Research*, 10:323–352, 1999.
16. M. J. Schoppers. Universal plans for Reactive Robots in Unpredictable Environments. In *Proc. IJCAI'87*, 1987.

OBDD-Based Optimistic and Strong Cyclic Adversarial Planning

Rune M. Jensen, Manuela M. Veloso and Michael H. Bowling
Computer Science Department,
Carnegie Mellon University,
5000 Forbes Ave, Pittsburgh,
PA 15213-3891, USA
Email: {runej,mmv,mhb}@cs.cmu.edu
Tel.: +1 (412) 268-{3053,1474,3069}
Fax: +1 (412) 268-4801

Abstract

Recently, universal planning has become feasible through the use of efficient symbolic methods for plan generation and representation based on reduced ordered binary decision diagrams (OBDDs). In this paper, we address adversarial universal planning for multi-agent domains in which a set of uncontrollable agents may be adversarial to us (as in e.g. robotics soccer). We present two new OBDD-based universal planning algorithms for such adversarial non-deterministic finite domains, namely *optimistic adversarial planning* and *strong cyclic adversarial planning*. We prove and show empirically that these algorithms extend the existing family of OBDD-based universal planning algorithms to the challenging domains with adversarial environments. We further relate adverserial planning to positive stochastic games by analyzing the properties of adversarial plans when these are considered policies for positive stochastic games. Our algorithms have been implemented within the Multi-agent OBDD-based Planner, UMOP, using the Non-deterministic Agent Domain Language, *NADL*.

Keywords: adversarial universal planning, multi-agent planning, non-deterministic domains, stochastic games.

1 Introduction

Universal planning, as originally developed by Schoppers (1987), is an approach for handling environments with contingencies. Universal planning is particularly appealing for active environments causing actions to be non-deterministic. A universal plan associates each possible world state with actions relevant in that state for achieving the goal. Due to the non-deterministic outcome of actions, a universal plan is executed by iteratively observing the current state and executing an action in the plan associated with that state.

In the general case the non-determinism forces a universal plan to cover all the domain states. Since planning domains traditionally have large state spaces, this constraint makes the representation and generation of universal plans nontrivial. Recently,

reduced ordered binary decision diagrams (OBDDs,[1]) have been shown to be efficient for synthesizing and representing universal plans [2, 3, 6]. OBDDs are compact representations of Boolean functions that have been successfully applied in *symbolic model checking* [7] to implicitly represent and traverse very large state spaces. Using similar techniques, a universal plan represented as an OBDD can be efficiently synthesized by a backward breadth-first search from the goal to the initial states in the planning domain.[1]

There are three OBDD-based planning algorithms: *strong*, *strong cyclic* and *optimistic planning*. Strong and strong cyclic planning were contributed within the MBP planner [2, 3]. MBP specifies a planning domain in the action description language $\mathcal{AR}$ [4]. The strong planning algorithm tries to generate a strong plan, i.e., a plan where all possible outcomes of the actions in the plan change the state to a new state closer to the goal. The strong cyclic planning algorithm returns a strong plan, if one exists, or otherwise tries to generate a plan that may contain loops but is guaranteed to achieve the goal, given that all cyclic executions eventually terminate.

Optimistic planning was contributed within the UMOP planner [6]. UMOP specifies a planning domain in the non-deterministic agent domain language, *NADL* , that explicitly defines a controllable system and an uncontrollable environment as two sets of synchronous agents. Optimistic planning tries to generate a relaxed plan where the only requirement is that there exists an outcome of each action that leads to a state nearer the goal.

None of the previous algorithms are generally better than the others. Their strengths and limitations depend on the structure of the domain [6]. However, a limitation of the previous algorithms is that they can not reason explicitly about environment actions, due to their usage of an *implicit* representation of the effect of these actions. [2] This is an important restriction for adversarial domains, as for the strong cyclic and optimistic algorithms, an adversarial environment may be able to prevent goal achievement.

In this paper we contribute two new planning algorithms robust for adversarial environments: *optimistic adversarial planning* and *strong cyclic adversarial planning*. These algorithms explicitly represent environment actions. The planner can then reason about these actions and take adversarial behavior into account. We prove that, in contrast to strong cyclic plans, strong cyclic *adversarial* plans guarantee goal achievement independently of the environment behavior. Similarly, we prove that optimistic adversarial plans improve the quality of optimistic plans by guaranteeing that a goal state can be reached from any state covered by the optimistic adversarial plan independently of the behavior of the environment. The results are verified empirically in a scalable example domain using an implementation of our algorithms in the UMOP planning framework.

Adversarial planning is related to game theory. The main difference is that the goal is represented in terms of a set of states instead of a utility function. Unlike strong cyclic adversarial planning, game tree algorithms, such as alpha-beta mini-max [5], can only guarantee goal achievement if the search is complete and the opponent uses a strict mini-max strategy. In practice, though, the explicit-state search has to be depth-bounded which reduces the approach to heuristic action selection. Matrix games are stateless and therefore strictly less expressive. The game-theoretic framework that is closest in relation to adversarial planning is stochastic games (SGs). Stochastic games extend Markov decision processes (MDPs) to multiple agents. An MDP has transition probabilities and is thus more expressive than the non-deterministic transition model of adversarial planning. However, by translating an adversarial planning problem into an SG problem by adding non-zero transition probabilities, we prove that an optimistic adversarial plan exists if and only if the solution to the corresponding positive stochastic game has a positive expected reward. Moreover, if a strong cyclic adversarial plan

[1] This work assumes that the non-deterministic domain definition is known and the focus is on the development of effective universal planning algorithms under this assumption.

[2] Figure 1(b) illustrates this restricted representation for an example domain introduced in next section.

exists, then the solution to the corresponding stochastic game has a maximum expected reward.

The restricted domain model of adversarial planning is suitable for problems where transition probabilities are irrelevant (e.g. worst case analysis). The advantage of this domain model compared to the MDP model of SGs is that it allows the application of efficient OBDD-based symbolic solution methods that potentially scale to much larger domains than can be handled by the explicit-state value iteration methods (e.g. [9]) used for solving stochastic games.

The remainder of the paper is organized as follows. Section 2 defines our representation of adversarial domains and introduces an example domain used throughout the paper. Section 3 defines the optimistic and strong cyclic adversarial planning algorithms and proves their key properties. In Section 4 we define and prove properties of the stochastic game representation of the adversarial planning problems. Finally, Section 5 draws conclusions and discusses directions for future work.

2 Adversarial Plan Domain Representation

An *NADL* domain[3] description consists of: a definition of *state variables*, a description of *system* and *environment agents*, and a specification of *initial* and *goal conditions*. The set of state variable assignments defines the state space of the domain. An agent's description is a set of *actions*. An action has a precondition and an effect defining in which states the action is applicable and what states the action can lead to. At each step, all of the agents perform exactly one action, and the resulting action tuple is a *joint action*.[4] The system agents model the behavior of the agents controllable by the planner, while the environment agents model the uncontrollable environment. There are two causes of non-determinism in *NADL* domains: (1) actions having multiple possible outcomes, and (2) uncontrollable concurrent environment actions. System and environment actions are assumed to be independent, such that an action chosen by the system in some state can not influence the set of actions that can be chosen by the environment in that state and vice versa. No assumptions are made about the behavior of environment agents. They might be *adversarial*, trying to prevent goal achievement of the system agents.

We represent the transition relation of an *NADL* domain with a Boolean function, $T(S, a_s, a_e, S')$. S is the current state, a_s and a_e are system and environment actions and S' is a next state. $T(S, a_s, a_e, S')$ is true if and only if S' is a *possible* next state when executing (a_s, a_e) in S.

A *planning problem* is a tuple (T, I, G), where T is a transition relation, and I and G are Boolean functions representing the initial and goal states, respectively. A *universal plan*, U, is a partial mapping from the domain states to the power set of system actions. A universal plan would be *executed* by the system agents by iteratively observing the current state and executing one of the actions in the universal plan associated to that state.

As an example, consider the domain shown in Figure 1. This domain has a single system agent that can execute the actions $+s$ and $-s$, and a single environment agent that can execute the actions $+e$ and $-e$. Edges in the figure are labeled with the corresponding joint action. There are 5 states, namely I, F, D, U and G. I and G are the only initial and goal states, respectively. D is a dead end state, as the goal is unreachable from D. This introduces an important difference between F and U, that captures a main aspect of the adversarial planning problem. We can view the two states F and U as states in which the system and the environment have different opportunities. Observe that the system "wins", i.e., reaches the goal, only if the signs of the two actions

[3] The reader is referred to [6] for a formal definition of *NADL*.

[4] In the remainder of the paper we will often refer to joint-actions as simply actions.

in the joint action are different. Otherwise it "loses", as there is no transition to the goal with a joint action with actions with the same sign. The goal is reachable from both states F and U. However the result of a "losing" joint action is different for F and U. In F, the system agent remains in F. Thus, the goal is still reachable for a possible future action. In U, however, the agent may transition to the dead end state D, which makes it impossible to reach the goal in subsequent steps.

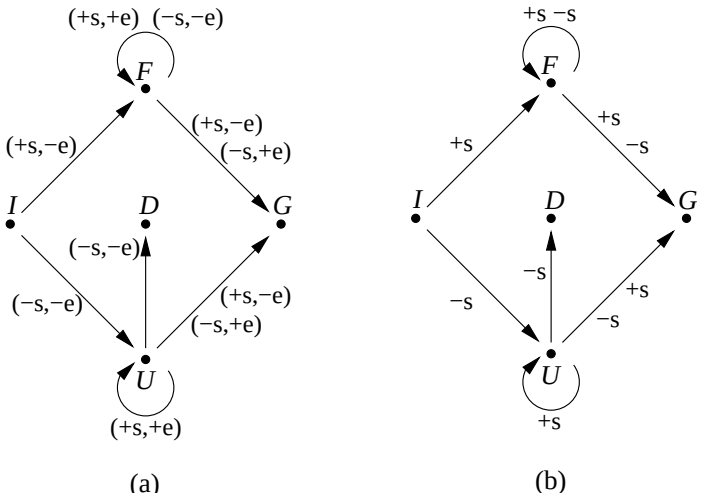


Figure 1: An adversarial planning domain example with initial state I and goal state G. There is a single system and environment agent with actions $\{+s, -s\}$ and $\{+e, -e\}$, respectively. (a) shows the explicit representation of environment actions used by our adversarial planning algorithms, while (b) shows the implicit representation used by previous algorithms, where the effect of environment actions is modeled by the nondeterminism of system actions.

Now consider how an adversarial environment can take advantage of the possibility of the system reaching a dead end from U. Since the system may end in D, when executing $-s$ in U, it is reasonable for the environment to assume that the system will always execute $+s$ in U. But now the environment can prevent the system from ever reaching the goal by always choosing action $+e$, so the system should completely avoid the path through U.

This example domain is important as it illustrates how an adversarial environment can act purposely to obstruct the goal achievement of the system. We will use it in the following sections to explain our algorithms. A universal plan, guaranteeing that G is eventually reached, is $\{(I, \{+s\}), (F, \{+s, -s\})\}$. In contrast to any previous universal planning algorithm, the strong cyclic adversarial planning algorithm can generate this plan as shown in Section 3.3.

3 Adversarial Planning

We introduce a generic function *Plan*(T, I, G) for representing OBDD-based universal planning algorithms. The algorithms, including ours, only differ by definition of the function computing the precomponent (*PreComp*(T, V)).

The generic function performs a backward breadth-first search from the goal states to the initial states. In each step the precomponent, U_p, of the visited states, V, is

computed. The precomponent is a partial mapping from states to the power set of system actions. Each state in the precomponent is mapped to a set of relevant system actions for reaching V.

function *Plan*(T, I, G)
  $U := \emptyset$; $V := G$
  while $I \not\subset V$
    $U_p := PreComp(T, V)$
    if $U_p = \emptyset$ **then return** *failure*
    else $U := U \cup U_p$
      $V := V \cup states(U_p)$
  return U

If the precomponent is empty a fixpoint of V has been reached that does not cover the initial states. Since this means that no universal plan can be generated that covers the initial states, the algorithm returns *failure*. Otherwise, the precomponent is added to the universal plan and the states in the precomponent are added to the set of visited states. All sets and mappings in the algorithm are represented by OBDDs. In particular, the universal plan and the precomponent are represented by the characteristic function of the set of state-actions pairs in the mapping.

3.1 The Optimistic Adversarial Precomponent

The core idea in adversarial planning is to be able to generate a plan for the system agents that ensures that the environment agents, even with complete knowledge of the domain and the plan, are unable to prevent reaching the goals. We formalize this idea in the definition of a *fair state*. A state s is fair with respect to a set of states, V, and a universal plan, U, if, for each applicable environment action, there exists a system action in U such that the joint action may lead into V. Let $A_T(s)$ denote the set of applicable environment actions in state s for transition system T. We then formally have:

Definition 1 (Fair State) *A state, s, is fair with respect to a set of states, V, and a universal plan, U, if and only if* $\forall a_e \in A_T(s) \,.\, \exists a_s \in U(s) \,.\, T(s, a_s, a_e, s') \wedge s' \in V$.

For convenience we define an *unfair* state to be a state that is not fair. The optimistic adversarial precomponent is an optimistic precomponent pruned for unfair states. In order to use a precomponent for OBDD-based universal planning, we need to define it as a boolean function represented by an OBDD. The optimistic adversarial precomponent (OAP) is the characteristic function of the set of state-action pairs in the precomponent:

Definition 2 (OAP) *Given a transition relation, T, the optimistic adversarial precomponent of a set of states, V, is the set of state-action pairs given by the characteristic function:*

$$OAP(T,V)(s,a_s) = \big(\forall a_e \,.\, A(T)(s,a_e) \Rightarrow J(T,V)(s,a_e)\big) \wedge \quad (1)$$

$$\big(\exists a_e, s' \,.\, T(s,a_s,a_e,s') \wedge s' \in V \wedge s \notin V\big) \quad (2)$$

$$J(T,V)(s,a_e) = \exists a_s, s' \,.\, T(s,a_s,a_e,s') \wedge s' \in V \quad (3)$$

$$A(T)(s,a_e) = \exists a_s, s' \,.\, T(s,a_s,a_e,s'). \quad (4)$$

Line (1) ensures that the state is fair. It says that, for any state in the precomponent, every applicable environment action (defined by $A(s, a_e)$) must also be included in a joint action leading to V (defined by $J(s, a_e)$). Line (2) says that every system action a_s relevant for a state $s \notin V$ must have at least one next state in V.

Figure 2 shows the optimistic adversarial precomponent of state G for the example domain ($OAP(T, G) = \{(F, +s), (F, -s), (U, +s), (U, -s)\}$). For clarity we include the transitions of the actions in the precomponent.

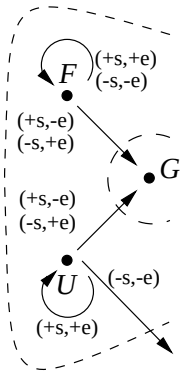


Figure 2: The OAP of G for the example of Figure 1.

3.2 The Strong Cyclic Adversarial Precomponent

A strong cyclic adversarial plan is a strong cyclic plan where every state is fair. Thus, all the actions in the plan lead to states covered by the plan. In particular, it is guaranteed that no dead ends are reached. The strong cyclic adversarial precomponent (SCAP) consists of a union of optimistic precomponents where each state in one of the optimistic precomponents is fair with respect to all states in the previous precomponents and the set of visited states.

The algorithm for generating an SCAP adds one optimistic precomponent at a time. After each addition, it first prunes actions with possible next states not covered by the optimistic precomponents and the set of visited states. It then subsequently prunes all the states that are no longer fair after the pruning of outgoing actions. If all the states are pruned from the precomponent, the algorithm continues adding optimistic precomponents until no actions are outgoing. Thus, in the final SCAP, we may have cycles due to loops and transitions crossing the boundaries of the optimistic precomponents. Again, we define the precomponent as the characteristic function of a set of state-action pairs:

Definition 3 (SCAP) *Given a transition relation, T, the strong cyclic adversarial precomponent of a set of states, V, is the set of state-action pairs given by the characteristic function $SCAP(T, V)(s, a_s)$.*

function $SCAP(T, V)$
  $i := 0$; $OA_0 := OAP(T, V)$
  while $OA_i \neq \emptyset$
    $SCA := PruneSCA(T, V, \mathbf{OA}, i)$
    if $SCA \neq \emptyset$ **then**
      return SCA
    else $i := i + 1$
      $OA_i := OAP\big(T, V \cup (\bigcup_{k=0}^{i-1} states(OA_k))\big)$
  return $\emptyset$

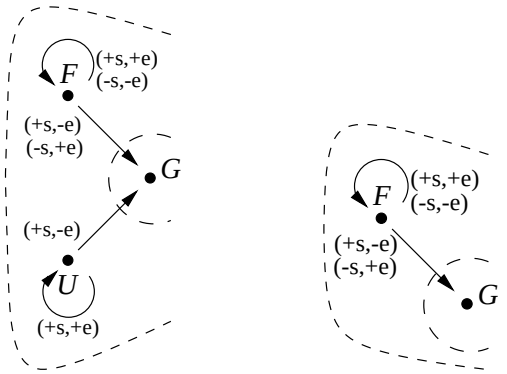


Figure 3: (a) The OAP pruned for actions with outgoing transitions; (b) The SCAP of G, for the example of Figure 1.

function $PruneSCA(T, V, \mathbf{OA}, i)$
  repeat
    $SCA := \cup_{k=0}^{i} OA_k$
    $V_C := V$; $V_T := V \cup \left(\bigcup_{k=0}^{i} states(OA_k) \right)$
    for $j = 0$ **to** i
      $P := PruneOutgoing(T, V_T, OA_j)$
      $OA_j := PruneUnfair(T, V_C, P)$
      $V_C := V_C \cup states(OA_j)$
  until $SCA = \cup_{k=0}^{i} OA_k$
  return SCA

$$PruneOutgoing(T, V, OA) = OA(s, a_s) \wedge \left(\forall a_e, s' \,.\, T(s, a_s, a_e, s') \Rightarrow V(s')\right)$$
$$PruneUnfair(T, V, OA) = OA(s, a_s) \wedge \left(\forall a_e \,.\, A(T)(s, a_e) \Rightarrow J(T, V)(s, a_e)\right)$$

$SCAP(T, V)$ adds an optimistic adversarial precomponent until the pruning function $PruneSCA(T, V, \mathbf{OA}, i)$ returns a non-empty precomponent. The pruning function keeps a local copy of the optimistic adversarial precomponents in an array $\mathbf{OA}$. SCA is the precomponent found so far. The pruning continues until SCA reaches a fix point. V_T is the set of states in the current precomponent. In each iteration transitions leading out of V_T are pruned. States turning unfair with respect to V_C, because of this pruning, are then removed. V_C is the union of all the states in the previous optimistic precomponents and the set of visited states V.

For an illustration, consider again the OAP of G (see Figure 2). Action $-s$ would have to be pruned from U since it has an outgoing transition. The pruned OAP is shown in Figure 3(a). Now there is no action leading to G in U when the environment chooses $+e$. U has become unfair and must be pruned from the precomponent. Since the precomponent is non-empty no more optimistic precomponents have to be added. The resulting precomponent, $SCAP(G) = \{(F, +s), (F, -s)\}$, is shown in Figure 3(b).

3.3 Properties of the Algorithms

Optimistic and strong cyclic adversarial planning extend the previous OBDD-based universal planning algorithms by pruning unfair states from the plan. For example, for the domain of Figure 1, the strong cyclic planning algorithm would generate the plan $\{(I, \{+s, -s\}), (F, \{+s, -s\}), (U, \{+s\})\}$, while the strong cyclic adversarial planning algorithm, as introduced above, generates the plan $\{(I, \{+s\}), (F, \{+s, -s\})\}$. It is capable of pruning U from the plan, since it becomes unfair. Also note that the plan is not a plain strong plan since progress towards the goal is not guaranteed in every state. It is easy to verify that there actually is no strong plan for this domain.

In order to state formal properties of adversarial planning, we define the *level* of a state to be the number of optimistic adversarial precomponents from the goal states to the state. We can now prove the following theorem:

Theorem 1 (SCA Termination) *The execution of a strong cyclic adversarial plan eventually reaches a goal state if it is initiated in some state covered by the plan and actions in the plan are chosen randomly.*

Proof: Since all unfair states and actions with transitions leading out of the strong cyclic adversarial plan have been removed, all the visited states will be fair and covered by the plan. From the random choice of actions in the plan it then follows that each execution of a plan action has a non-zero probability of leading to a state at a lower level. Let m be the maximum level of some state in the strong cyclic adversarial plan (m exists since the state space is finite). Let p denote the smallest probability of progressing at least one level for any state in the plan. Then, from every state in the plan, the probability of reaching a goal state in m steps is at least p^m. Thus, the probability of reaching a goal state in mn steps is at least $1 - (1 - p^m)^n$, which approaches 1 for $n \to \infty$. Thus, a goal state will eventually be reached. □

The termination theorem makes strong cyclic adversarial plans more powerful than strong cyclic plans since termination of strong cyclic plans only can be guaranteed by assuming no infinite looping (i.e. a "friendly" environment). For optimistic adversarial plans, termination can not be proved since dead ends may be reached. However, since optimistic plans only consist of fair states, there is a chance of progressing towards the goal in each state:

Theorem 2 (OA Progress) *The execution of an optimistic adversarial plan has a non-zero probability of eventually reaching the goal from each state covered by the plan if the actions in the plan are chosen randomly.*

Proof: This follows directly from the fact that each state in the plan is fair. □

Optimistic plans do not have this property since unfair states may be included in the plans. Thus, it may be possible for the environment to prevent the system from progressing towards the goal either by forcing a transition to a dead end or by making transitions cyclic.

3.4 Empirical Results

The adversarial and previous OBDD-based universal planning algorithms have been implemented in the publicly available UMOP planning framework. In order to illustrate the scalability of our approach, we use a general version of the example domain of Figure 1, as shown in Figure 4.

The domain has a single system and environment agent with actions $\{+s, -s, l\}$ and $\{+e, -e\}$, respectively. The system progresses towards the goal if the signs of the two actions in the joint action are different. At any time, it can switch from the lower

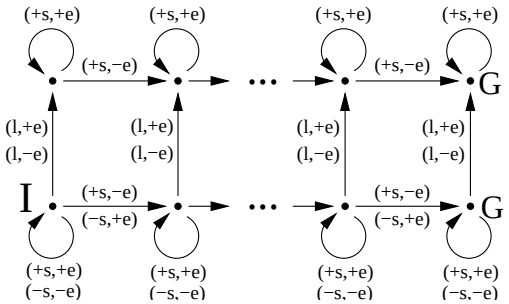


Figure 4: A general version of the domain shown in Figure 1.

to the upper row of states by executing l. In the upper row, the system can only execute $+s$. Thus, in these states an adversarial environment can prevent further progress by always executing $+e$.

Figure 5 shows, in a double logarithmic scale, the running time and the plan size as a function of the number of domain states when finding strong cyclic and strong cyclic adversarial plans.[5] In this domain both algorithms scale well. The experiment indicate a polynomial time complexity for both algorithms. For the largest domain with 65,536 states the CPU time is less than 32 seconds for generating the strong cyclic adversarial plan. The results also demonstrate the efficiency of OBDDs for representing universal plans in structured domains. The largest plan consists of only 38 OBDD nodes.

The strong cyclic adversarial plans only consider executing $-s$ and $+s$, while the strong cyclic plans consider all applicable actions. Hence, the strong cyclic adversarial plans have about twice as many OBDD nodes and take about twice as long time to generate. But this slightly higher cost of generating a strong cyclic adversarial plan pays off well in plan quality. The strong cyclic adversarial plan is guaranteed to achieve the goal when choosing actions in the plan randomly. In contrast, the probability of achieving the goal in the worst case for the strong cyclic plan is less than $(\frac{2}{3})^{N/2-1}$, where N is the number of states in the domain. Thus, for an adversarial environment the probability of reaching the goal with a strong cyclic plan is practically zero, even for small instances of the domain.

4 Relation to Stochastic Games

A *stochastic game* is a tuple $(n, S, A_{1\ldots n}, T, R_{1\ldots n})$, where n is the number of agents, S is the set of states, A_i is the set of actions available to player i (and A is the joint action space $A_1 \times \ldots \times A_n$), T is the transition function $S \times A \times S \rightarrow [0, 1]$, and R_i is a reward function for the ith agent $S \rightarrow \mathcal{R}$. A solution to a stochastic game is a stationary and possibly stochastic policy, $\rho : S \rightarrow PD(A_i)$, for an agent in this environment that maps states to a probability distribution over its actions. The goal is to find such a policy that maximizes the agent's future discounted reward. In a zero-sum stochastic game, two agents have rewards adding up to a constant value, in particular, zero. The value of a discounted stochastic game with discount factor γ is a vector of

[5]The experiment was carried out on a 500 MHz Pentium III PC with 512 MB RAM running Red Hat Linux 5.2.

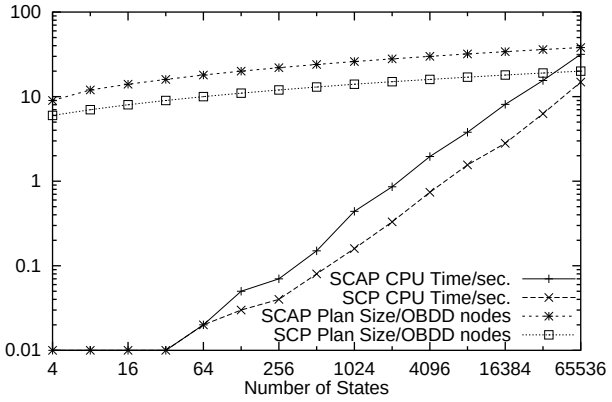


Figure 5: CPU time and plan size of strong cyclic and strong cyclic adversarial plans as a function of domain size.

values, one for each state, satisfying the equation:

$$V_\gamma(s) \quad = \quad \max_{\sigma \in PD(A_s)} \min_{a_e \in A_e} \sum_{a_s \in A_s} \sigma(a_s) \left(\sum_{s' \in S} T(s, a_s, a_e, s') \left(R(s') + \gamma V_\gamma(s') \right) \right)$$

For positive stochastic games, the payoffs are summed without a discount factor, which can be computed as $V(s) = \lim_{\gamma \to 1^-} V_\gamma(s)$.

The derived stochastic game from a universal planning problem is given by the following definition:

Definition 4 *A* derived stochastic game *from a planning problem, (T, I, G), is a zero-sum stochastic game with states and actions identical to those of the planning problem. Reward for the system agent is one when entering a state in G, zero otherwise. Transition probabilities, $\bar{T}$, are any assignment satisfying,*

$$\begin{array}{ll} \bar{T}(s, a_s, a_e, s') \in (0, 1] & \textit{if } T(s, a_s, a_e, s') \\ \bar{T}(s, a_s, a_e, s') = 0 & \textit{otherwise} \end{array}$$

We can now prove the following two theorems:

Theorem 3 *An optimistic adversarial plan exists if and only if, for* **any** *derived stochastic game, the value at all initial states is positive.*

Proof: ($\Rightarrow$) We prove by induction on the level of a state. For a state, s, at level one, we know the state is fair with respect to the goal states. So, for every action of the environment, there exists an action of the system that transitions to a goal state with some non-zero probability. If T_{min} is the smallest transition probability, then if the system simply randomizes among its actions, it will receive a reward of one with probability $\frac{T_{min}}{|A_s|}$. Therefore, $V(s) \geq \frac{T_{min}}{|A_s|} > 0$. Consider a state, s, at level n. Since s is fair, we can use the same argument as above that the next state will be at a level less than n with probability $\frac{T_{min}}{|A_s|}$. With the induction hypothesis, we get,

$$V(s) \geq \frac{T_{min}}{|A_s|} V(s') > 0$$

Therefore, all states in the set V have a positive value. Since the algorithm terminates with $I \subseteq V$, then all initial states have a positive value.
($\Leftarrow$) Assume for some derived stochastic game that the value at all initial states is positive. Consider running the optimistic adversarial planning algorithm and for, the purpose of contradiction, assume the algorithm terminates with $I \not\subseteq V$. Consider the state $s^* \notin V$ that maximizes $V_\gamma(s^*)$. We know that, since the algorithm terminated, s^* must not be fair with respect to V. So there exists an action of the environment, a_e, such that, for all a_s, the next state will not be in V. So we can bound the value equation by assuming the environment selects action a_e,

$$V_\gamma(s^*) \quad \leq \quad \max_\sigma \sum_{a_s \in A_s} \sigma(a_s) \left(\sum_{s' \notin V} T(s^*, a_s, a_e, s') \left(\gamma V_\gamma(s')\right) \right)$$

Notice that we do not have to sum over all possible next states since we know the transition probabilities to states in V are zero (by the selection of a_e). We also know that immediate rewards for states not in V are zero, since they are not goal states. By our selection of s^* we know that $V_\gamma(s')$ must be smaller than the value of s^*. By substituting this into the inequality we can pull it outside of the summations which now sum to one. So $V(s^*) = 0$, as we need to satisfy:

$$V_\gamma(s^*) \leq \gamma V_\gamma(s^*) \cdot 1$$

Since any initial state is not in V, $V(s^*)$ must have value equal to zero, which is a contradiction to our initial assumption. So, the algorithm must terminate with $I \subseteq V$, and therefore an optimistic adversarial plan exists. □

Theorem 4 *If a strong cyclic adversarial plan exists, then for* **any** *derived stochastic game, the value at all initial states is 1.*

Proof: Consider a policy π that selects actions with equal probability among the unpruned actions of the strong cyclic adversarial plan. For states not in the adversarial plan select an action arbitrarily. We will compute the value of executing this policy, V_γ^π.

Consider a state s in the strong cyclic adversarial plan such that $V_\gamma^\pi(s)$ is minimal. Let $L(s) = N$ be the level of this state. We know that this state is fair with respect to the states at level less than N, and therefore (as in Theorem 1) the probability of reaching a state s' with $L(s') \leq N - 1$ when following policy π is at least $p = \frac{T_{min}}{|A_s|} > 0$. This same argument can be repeated for s', and so after two steps, with probability at least p^2, the state will be s'' where $L(s'') \leq L(s') - 1 \leq N - 2$. Repeating this, after N steps with probability p^N, the state will be s^N where $L(s^N) \leq L(s) - N \leq 0$, so s^N must be a goal state and the system received a reward of one.

So we can consider the value at state s when following the policy π. We know after N steps if it has not reached a goal state it must be in some state still in the adversarial plan due to the enforcement of the strong cyclic property. In essence, all actions with outgoing transitions are pruned and therefore are never executed by π. The value of this state by definition must be larger than $V_\gamma^\pi(s)$. Therefore,

$$\begin{aligned} V_\gamma^\pi(s) &\geq p^N \gamma^N \cdot 1 + (1 - p^N)\gamma^N V_\gamma^\pi(s) \\ &\geq \frac{\gamma^N p^N}{1 - (1 - p^N)\gamma^N} \\ \lim_{\gamma \to 1} V_\gamma^\pi(s) &\geq \frac{p^N}{1 - (1 - p^N)} \geq 1 \end{aligned}$$

So $V^\pi(s) = 1$ and since s is the state with minimal value, for any initial state s_i, $V^\pi(s_i) = 1$. Since $V(s_i)$ maximizes over all possible policies, $V(s_i) = 1$. □

5 Conclusion

This paper contributes two new OBDD-based universal planning algorithms, namely optimistic and strong cyclic adversarial planning. These algorithms naturally extend the previous optimistic and strong cyclic algorithms to adversarial environments. We have proved that, in contrast to optimistic plans, optimistic adversarial plans always have a non-zero probability of reaching a goal state. Furthermore, we have proved and shown empirically that, in contrast to strong cyclic plans, a strong cyclic adversarial plan always eventually reach the goal. Finally, we introduced and proved several relations between adversarial universal planning and positive stochastic games. An interesting direction for future work is to investigate if adversarial planning can be used to scale up the explicit-state approaches for solving stochastic games by pruning states and transitions irrelevant for finding an optimal policy.

Acknowledgments

We wish to thank Scott Lenser for early discussions on this work. The research is sponsored in part by the Danish Research Agency and the United States Air Force under Grants Nos F30602-00-2-0549 and F30602-98-2-0135. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force, the Danish Research Agency, or the US Government.

References

[1] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[2] A. Cimatti, M. Roveri, and P. Traverso. OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of AAAI-98*, pages 875–881. AAAI Press, 1998.

[3] A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS'98)*, pages 36–43. AAAI Press, 1998.

[4] E. Giunchiglia, G. Kartha, and Y. Lifschitz. Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95:409–438, 1997.

[5] T. P. Hart and D. J. Edwards. The tree prune (TP) algorithm. Technical report, MIT, 1961. Artificial Intelligence Project Memo 30.

[6] R. Jensen and M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, 13:189–226, 2000.

[7] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[8] M. Schoppers. Universal planning for reactive robots in unpredictable environments. In *Proceedings of IJCAI-87*, pages 1039–1046, 1987.

[9] L. S. Shapley. Stochastic games. *PNAS*, 39:1095–1100, 1953.

Multi-Agent Off-line Coordination: Structure and Complexity

Carmel Domshlak and Yefim Dinitz

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
{dcarmel,dinitz}@cs.bgu.ac.il

Abstract. Coordination between processing entities is one of the most widely studied areas in multi-agent planning research. Recently, efforts have been made to understand the formal computational issues of this important area. In this paper, we make a step toward this direction, and analyze a restricted class of coordination problems for dependent agents with independent goals acting in the same environment. We assume that a state-transition description of each agent is given, and that preconditioning an agent's transitions by the states of other agents is the only considered kind of inter-agent dependence. Off-line coordination between the agents is considered. We analyze some structural properties of these problems, and investigate the relationship between these properties and the complexity of coordination in this domain. We show that our general problem is provably intractable, but some significant subclasses are in NP and even polynomial.

1 Introduction

Coordination between processing entities is one of the most widely studied areas in multi-agent planning research. Recently, efforts have been made to understand the formal computational issues in this important area. In this paper, we make a additional step toward this direction. We describe a restricted class of coordination problems for agents with independent goals acting in the same environment, that is the strategy of each agent should be planned while taking into account strategies of other agents as planning constraints. In these problems, (i) the set of feasible plans for each agent is defined by a state-transition graph, and (ii) preconditioning an agent's *transitions* by the *states* of other agents is the only considered kind of inter-agent dependence. For this problem class, off-line coordination is considered: structural and computational properties are investigated, and both tractable and intractable subclasses are presented. Although the examined class of problems can be seen as significantly restricted, its place in multi-agent systems was already discussed in AI literature, e.g. [6, 14, 22]. We believe that the formal model we suggest for the presented problem class can serve as a basis for further extensions toward a representation of richer multi-agent worlds yet preserving convenience for computational analysis.

In the area of multi-agent coordination, we identify two main research directions: *multi-agent collaboration* and *synthesizing multi-agent plans*. In multi-agent collaboration, goal achievement is a team activity, i.e., multiple agents collaborate towards the

achievement of an overarching goal. During the last decade, various theories for multi-agent collaboration, e.g. [9, 18, 20], and several generic approaches for corresponding planning, e.g. [11, 25], were proposed; for an overview of this area see [17]. In addition, a few formal computational results concerning the coalition formation are found for both benevolent [24] and autonomous [12] agents.

Synthesizing multi-agent plans addresses synchronization between agents that work on independent goals in the same environment. In [16], centralized merging of preconstructed individual agent plans is presented, and it is based on introducing synchronization actions. In [15], a heuristic approach that exploits decomposed search space is introduced. Whereas in [15, 16] methods for merging plans at the primitive level are described, in [8] a strategy for plan merge on different levels of hierarchical plans is presented. In [1, 2], a generic approach for distributed, incremental merging of individual plans is suggested: the agents are considered sequentially, and the plan for the currently processed agent is synchronized with the plans chosen for the previously processed agents. Besides, a few more algorithmic approaches to an application-specific plan synchronization are proposed in [7, 23, 26]. Note that all these approaches for inter-agent synchronization assume that the personal plans are already constructed, and that the merging operation is considered only on this fixed set of individual plans (possibly, certain local modifications of the plans may be allowed). This essentially restricts not only the quality, but even the ability to find a coordinated plan.

A wider formal model for multi-agent systems that concerns both determining personal agent plans and their synchronization, is suggested in [22]. With respect to this model, the cooperative goal achievement problem is defined. It can be roughly stated as follows: Given a set of benevolent agents in the same environment, each one with its own, independent goal, does there exist a satisfying system of coordinated personal plans? In [22], this problem is shown to be PSPACE-complete.

In this paper, we concentrate on a particular class of the cooperative goal achievement problem. In this problem class, which we denote to as STS (state-transition support), an action (transition) of an agent can be constrained by the states of other agents. Following [6, 22], each agent is assumed to be representable as a finite automaton. As in [8, 15, 16], the coordination process is assumed to be off-line and centralized. Note that we are not dealing here with many complementary issues of multi-agent systems such as determining agent behavior, collaboration establishing, decision process distributing, etc.

We suggest to represent an STS problem domain using a graph language. The two interrelated graphical structures of our model are as follows:

1. The *strategy* (*di*)*graph*: the state-transition graph describing behavior of an agent.
2. The *dependence* (*di*)*graph*: description of the dependence between the agents.

The strategy graph compactly captures all alternative personal plans for the corresponding agent: its strategies correspond to the *paths* from the source node to the target node in its strategy graph. Transition of an agent along an edge is conditioned by the *states* of the agents on which it depends, i.e., their locations at certain nodes in their strategy graphs. The reader can find some relation between our strategy graphs and the interactive automata of [21]. The main differences are that we (i) address preconditioning, not interaction between agents, and (ii) consider off-line planning.

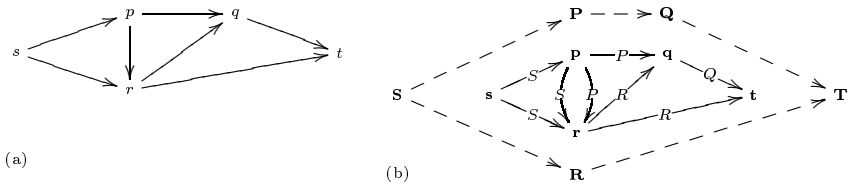


Fig. 1. Examples of strategy graphs for: (a) single agent, (b) two-agent group: transitions of X depend on states of Y.

The nodes of the dependence graph correspond to the agents. An edge (p, q) appears in the dependence graph if some action of the agent q depends on the state of the agent p. We define subclasses of STS, mainly with respect to structural properties of the dependence graph, and analyze their complexity. The complexity is measured in terms of the total size of the strategy graphs and the size of the dependence graph (which is actually the number of agents). We show that our general problem is provably intractable, but some significant subclasses are in NP and even polynomial. As far as we know, such an analysis of the relation between the structural and the computational properties of a multi-agent coordination problem was not done in previous research. Note that this paper presents only first results of our ongoing research, thus, in particular, we do not discuss relative significance of the presented problem classes, and we do not provide wide final conclusions.

The rest of the paper is organized as follows: Section 2 presents and discusses our model. Section 3 is devoted to the basic case of two agents. Section 4 introduces a classification of STS's subclasses, and presents additional complexity results. For the proofs and some of the detailed algorithms we refer to the technical report [13]. Concluding remarks are given in Section 5, together with some discussion of related issues and future work.

2 Graphical Model for STS

In this section we define two graphical structures of the STS problem domain. The *strategy graph* G^A captures the alternative personal strategies of an agent A. The agent's states are represented by the nodes of G^A. Each (directed) edge represents a possible transition between the two corresponding states. Multiple transitions between two states are possible. If a transition is conditioned, then the corresponding edge has a *label* which describes the condition; when a transition can be done under several alternative conditions, this is modeled by multiple edges, each labeled by a single condition. There are two unique nodes in G^A, the source node and the target node, which represent the initial and goal states of A, respectively. The possible agent's strategies are represented by the paths from the source node to the target node. For an illustration see Fig. 1(a), where the possible strategies are: $spqt$, $sprt$, $sprqt$, srt, $srqt$.

The size of this structure is linear in the length of the problem description. Observe that in general, the number of different, potentially applicable, strategies may be exponential in the size of the problem even for an acyclic structure; for a general structure, this number can, formally, be infinite.

Consider a pair of agents X and Y so that X depends on Y in the following sense: each transition of X is preconditioned by a certain state of Y. In graph terms, each edge in G^X is labeled by the name of a node in G^Y. An example is given in Fig. 1(b), where the inner graph models X and the outer one models Y. Note that there are two transitions of X from the state p to the state r which are preconditioned by different states of Y. The following pair of strategies for X and Y is coordinated: The agent X begins from state s and Y begins from state S. Subsequently, (i) X moves to p, (ii) Y moves to P, while X is in p, (iii) X moves to q, (iv) Y moves to Q, (v) X moves to t, and finally, (iv) Y moves to T. We denote this coordinated multi-agent strategy by the sequence $\langle (s,p)(S,P)(p,q)(P,Q)(q,t)(Q,T)\rangle$. The other possible coordinated plans for X and Y are $\langle (s,p)(p,r)(S,R)(r,t)(R,T)\rangle$ and $\langle (s,r)(S,R)(r,t)(R,T)\rangle$. No other coordinated pair of strategies exists. For example, if the agents begin with $\langle (s,p)(S,P)(p,r)\rangle$, then X cannot leave r, since it is impossible for Y to reach R after being in P.

In general, there are several agents $A_1, A_2, \ldots, A_n$ with strategy graphs $G^i = G^{A_i}$, $1 \leq i \leq n$. We say that an agent A_i depends on an agent A_j if transitions of A_i have states of A_j as preconditions. Note that a few agents may depend on the same agent, and an agent may depend on a few other agents. Hence, a transition of A_i is preconditioned by states of agents that A_i depends on; in general, a transition can depend on a part of them.

The *dependence graph* $\mathcal{G}$ is a digraph whose nodes are A_i, $1 \leq i \leq n$, and there is an edge from A_j to A_i if agent A_i depends on agent A_j. In what following, we identify nodes of $\mathcal{G}$ with the corresponding agents. Likewise, we identify the nodes and edges of G^i with the states and transitions of A_i, respectively. An edge of G^i has a compound label, each component of which is a state of an immediate predecessor of A_i in $\mathcal{G}$ (i.e., a node in the strategy graph of this predecessor).

Both the strategy graph and the dependence graph can be constructed straightforwardly, given an STS problem. Likewise, the structural properties of the dependence graph, which are investigated in this paper, can be verified in low polynomial time in the size of the graph.

In this paper, *each* transition of an agent is preconditioned by the states of *all* agents that this agent depends on, if the opposite is not declared explicitly. We consider only connected dependence graphs. The reason is that otherwise the set of agents can be divided into disjoint subsets that can be considered independently.

A natural motivation for a pair of agents X and Y as above is that X fulfills some mission, while Y supports it. In practice, the mission of X may be a movement towards a target, while Y either watches X from certain observation points, or feeds it from certain warehouses, or protects it from some fortified positions. Such a support is usual in military and is wide-spread in various other activities. For example, a worker X assembles some compound product, while a mobile crane Y moves the (half-assembled) product from one working place of X to another. In general, when there are several agents, each agent is supported by a few agents, and in turn, supports a few other agents.

Having read this far the reader may argue that it is possible that we never know an agent's behavior in complete detail, and even if we do, the number of states that

represent an agent's behavior may be very large. In this case, exploiting and even constructing strategy graphs seems to be infeasible.

Indeed, it is not realistic to present detailed models of many real-life systems as finite state machines. However, as it is argued in [22], the representation an agent uses need not present the world in sufficient detail to require more expressive description language. Intuitively, it is possible to distinguish between the physical state of agents and their computational state. It is the computational state that we have difficulty in modeling by finite-state machines. However, the computational state of an agent is not accessible to the other agents. On the other hand, a finite-state description can serve well as a representation of the physical state. A wider discussion of these and other issues of finite-automata-based knowledge representation and reasoning can be found in [22]. Likewise, number of finite-automata-based architectures are discussed in the AI literature, e.g., see [6].

In case that the number of parameters describing the physical state (and thus the size of the state space) is still large, it is often possible to take advantage of the structure of an agent in order to derive a concise description of its state. In particular, an agent may be viewed as being composed of a number of largely independent components, each with its own state. Each such component (such as robot's hand, motor, wheel, etc.) can be represented by an agent, whose physical state is relatively simple, since it describes a particular feature of a complex system. In this case, the strategy graphs of the agents are expected to be of reasonable size.

3 The Basic Case of Two Agents

In the rest of the paper, we study the complexity of various multi-agent coordination problems based on the model suggested above. In this section we demonstrate our techniques on solving the basic case of agent X depending on agent Y. Given the strategy graphs of X and Y, we build a new graph describing X restricted by Y. As a result, our problem is reduced to the known problem of finding a source-to-target path in this new graph. Such a reduction becomes possible because of the locality of the original restrictions.

We describe a solution of our problem as an interleaved sequence of transitions of agents X and Y, as was shown in Section 2. Let us analyze the structure of a general coordinated plan Π for these agents. As mentioned above, private strategies for X and Y are source-to-target paths in G^X and G^Y respectively, and we denote them by σ^X and σ^Y. Let us divide σ^X into intervals σ_1^X, σ_2^X, ..., σ_m^X of constancy of labels: all edges (transitions) in σ_i^X are labeled (preconditioned) by the same node (state) N_i of Y. The Y's path σ^Y is also divided by the nodes N_i into some parts. The considered plan is as follows: (1) If $N_1 \neq s^Y$, then agent Y moves from s^Y to state N_1 along the initial part of σ^Y; (2) Agent X moves along σ_1^X to its end; (3) Y moves from N_1 to N_2 along the next part of σ^Y; (4) X moves along σ_2^X, etc. Finally, if $N_m \neq t^Y$, then agent Y moves from N_m to t^Y along the final part of σ^Y. For illustration see Figure 2.

Let see what "degrees of freedom" Π has. For simplicity of notation, we denote s^Y by N_0 and t^Y by N_{m+1}. Observe that the replacement of part of σ^Y between N_i to N_{i+1}, $0 \leq i \leq m$, by any other path from N_i to N_{i+1} retains the plan feasible. That is,

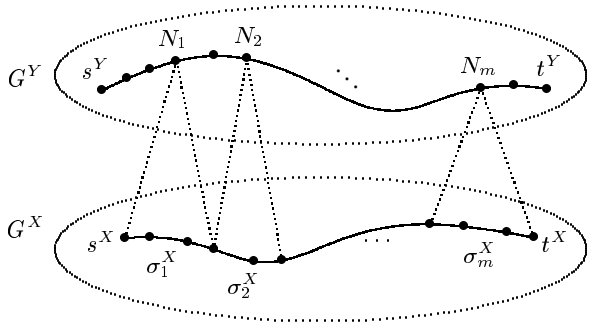


Fig. 2. Relation between personal strategies of X and Y

the only requirement for feasibility of σ^X is the *existence* of such a path for each label change along σ^X. Moreover, replacement of any σ_i^X by another path between its ends such that all its edges are labeled by N_i retains the plan feasible.

Lets now consider building such a coordinated plan Π. We take as a central issue finding an appropriate path σ^X. As mentioned above, the only critical points in its building are the moments of label change along σ^X: if the label changes from N_i to N_{i+1}, then a "connecting" path from N_i to N_{i+1} in G^Y must exist. In order to include into the same framework the cases in which the label of the first edge is not s^Y and the label of the last edge is not t^Y, we extend G^X as follows. We add to G^X a dummy source $\tilde{s}^X$, with an edge $(\tilde{s}^X, s^X)$ labeled s^Y, and a dummy target $\tilde{t}^X$, with an edge $(t^X, \tilde{t}^X)$ labeled t^Y, and consider path $\tilde{\sigma}^X$ from $\tilde{s}^X$ to $\tilde{t}^X$ in this extended graph $\tilde{G}^X$. It is easy to see that existence of connecting paths in G^Y for all critical points in $\tilde{\sigma}^X$ is necessary and sufficient for the feasibility σ^X. Indeed, the twin path σ^Y is the concatenation of all connecting paths, for $0 \leq i \leq m$. Therefore, the only information we need in order to build an appropriate path in $\tilde{G}^X$ is whether two edges may be consequent on a path $\tilde{\sigma}^X$ in the above sense, for all pairs of adjacent edges in $\tilde{G}^X$.

Information on existence of a path between any pair of nodes N' and N'' of G^Y (*reachability* of N'' from N') can be obtained by a preprocessing of G^Y. Appropriate algorithms are widely known and are very fast; the paths themselves, if any, are provided by these algorithms as well, e.g. see [10]. Hence, we can form the following database for all pairs of consequent edges in $\tilde{G}^X$: If an edge e' labeled N' enters some node and an edge e'' labeled N'' leaves the same node, then the pair (e', e'') is marked "permitted" if either $N' = N''$ or N'' is reachable from N' in G^Y. For example see Figure 3, where dotted lines show the permitted edge pairs.

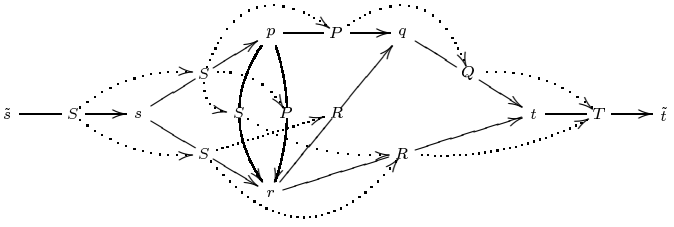


Fig. 3. Illustration of $\tilde{G}^X$ with the corresponding $G^{X|Y}$.

In fact, the suggested construction provides a reduction of the problem of finding a path $\tilde{\sigma}^X$ as defined above to a known problem of finding a path in a graph. Let the new graph $G^{X|Y}$ have the *edges* of $\tilde{G}^X$ as nodes, and let its edges be defined by the permitted pairs of edges in $\tilde{G}^X$.[1] Now, all we need is a path from the source node $(\tilde{s}^X, s^X)$ to the target node $(t^X, \tilde{t}^X)$ in $G^{X|Y}$. Clearly, such a path defines the corresponding path in $\tilde{G}^X$ immediately.

Figure 4 summarizes the algorithm for finding a coordinated plan for agent X depending on ("supported by") agent Y. This algorithm is, evidently, polynomial. The reader can "execute" it by himself on the example given in Figure 3. This example is the same as the one in Section 2, and any of the three feasible coordinated plans listed there can be obtained as the result of such an execution. Thus we have achieved our first result, and in what following, we use the presented approach for analyzing other variants of STS.

1. Extending strategy graph G^X to $\tilde{G}^X$.
2. Preprocessing of strategy graph G^Y: finding a path from any node to any other node (in fact, it is sufficient to do this for the node pairs needed in the next phase only).
3. Constructing permitted-edge graph $G^{X|Y}$.
4. Finding a source-to-target path in $G^{X|Y}$, if any, or reporting on non-existence of a coordinated plan, otherwise.
5. Reconstructing from the path found in phase 4 a path $\tilde{\sigma}^X$ in $\tilde{G}^X$ and its abridged variant σ^X in G^X.
6. Building a path σ^Y in G^Y as the concatenation of paths found at phase 2, for all label changes along $\tilde{\sigma}^X$.
7. Outputting properly interleaved strategies σ^X and σ^Y.

Fig. 4. Algorithm for the basic STS problem.

Theorem 1. *There exists a polynomial time algorithm that, given a pair of agents X, Y, in which X depends on Y, determines existence of a coordinated plan and finds such a plan, if exists.*

This result can be easily generalized to finding an *optimal* coordinated plan. Assume that the cost function is the sum or the product of weights of states and/or transitions used in the plan. Since in step 4 of the algorithm, an arbitrary path can be chosen, let us choose an optimal path; this suffices for global optimality. Any algorithm for finding a cheapest path in a graph can be used, e.g. see [10].

Now let us require, in addition, *simplicity* of Y's strategy, i.e., let us forbid Y to visit the same state twice. A motivation for this can be that each state of Y relates to some consumable resources. It turns out that this seemingly non-essential change in the problem definition, change the problem to be NP-complete. Informally, the reason for this complexity worsening is that the locality of precondition is thus broken. This result is mainly interesting from the theoretical point of view, since it points on a computational sensitivity of our original problem.

[1] Such a construction is a variant of the so called "edge graph" known in graph theory; the addition in our case is the exclusion of non-permitted edges from it.

4 Results for Some Other Subclasses of STS

In this section we discuss various additional complexity results for STS. Problem classes are defined mainly by the structure of their dependence graph. Moreover, both results and methods depend crucially on the number and size of possible agent strategies: if both of them are polynomial in the size of problem domain, then we distinguish such a class as *bounded*. Figure 5 summarizes the achieved results, and each box in the figure denotes an STS's subclass.

The notation is as follows: First, each STS subclass is denoted with respect to the form of the dependence graph: $\mathbb{C}$ stands for *directed chain*, $\mathbb{F}^{\wedge}$ for *directed fork* (graph with exactly one node with a non-zero outdegree), $\mathbb{F}^{\vee}$ for *directed inverse fork* (graph with exactly one node with a non-zero indegree), $\mathbb{P}$ for *polytree* (digraph whose underlying undirected graph is a tree), and $\mathbb{S}$ for *singly-connected DAG* (directed acyclic graph with at most one directed path between any pair of nodes). Second, the subscript denotes the number of agents: k stands for a constant bound, and n indicates no bounds.

By default, we assume that each transition of an agent is preconditioned by the states of *all* agents that this agent depends on. Alternatively, π in a superscript denotes a possibility of *partial* dependence. σ in a superscript denotes the requirement that all strategies chosen are *simple*. Note that cases $\mathbb{C}_2$ and $\mathbb{C}_2^{\sigma}$ have been discussed in Section 3.

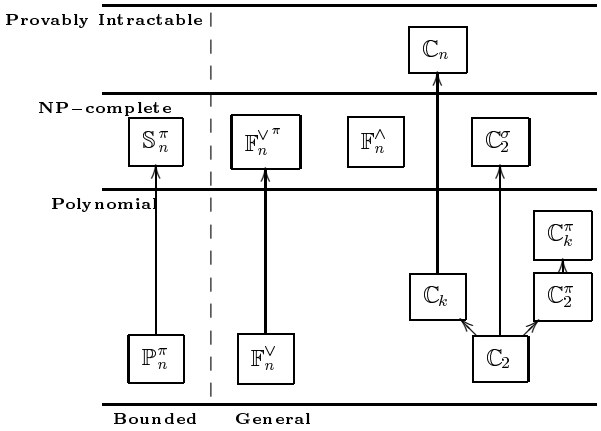


Fig. 5. Complexity results for STS

Coordinating General Agents A natural generalization $\mathbb{C}_n$ of the class $\mathbb{C}_2$ (as in Section 3) occurs when there is a dependence chain: agents A_i, $1 \leq i \leq n$, with A_i dependent on A_{i-1}, $i \geq 2$. For simplicity of notation, let us abridge A_i to i in superscripts (e.g., G^i instead of G^{A_i}).

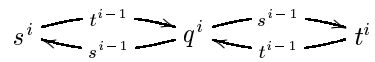
Our approach for $\mathbb{C}_n$ is to iterate the analysis and the algorithm for $\mathbb{C}_2$ along the dependence chain. A naive scheme is as follows. For processing pair A_3, A_2, we need a database on *coordinated* reachability in G^2. Observe that our algorithm for $\mathbb{C}_2$ applied to pair A_2, A_1 provides such an information for nodes s^2, t^2; clearly, this can be done

for any pair of nodes of G^2. Considering pair A_2, A_1, lets check reachability and store paths, if any, for *all* pairs of nodes in G^2, as required. Now, we are as if sure that if we would determine a coordinated plan for pair A_3, A_2 based on the above reachability relation for A_2, a corresponding strategy for A_1 will be obtained easily. Hence, the situation with A_3, A_2 is similar to that with A_2, A_1, and thus we can iterate up to A_n. Personal coordinated strategies can be consequently reconstructed, from A_{n-1} to A_1.

However, this scheme turns to be inconsistent. The point is that, in fact, the coordinated reachability relation in G^2 is on its *edges*, not on its nodes, and it is captured by the edge graph $G^{2|1}$. Therefore, informally, the right approach is to create an edge graph for G^3 basing on $G^{2|1}$, not on G^2. This idea of recursive edge graph creating can be generalized for dependence chain of arbitrary length. Using this method, we never arrive at a deadend as in the naive scheme, and we can obtain any coordinated plan for this problem. For the detailed algorithm for $\mathbb{C}_n$, together with an illustrating example of its execution and a counterexample for the naive approach we refer to the technical report [13].

Theorem 2. **(I)** *There exists an algorithm for $\mathbb{C}_n$ that determines existence of a coordinated plan and finds such a plan, if exists. This algorithm is polynomial in the total size of strategy graphs and exponential in n; thus $\mathbb{C}_k$ is proved to be polynomial.*
(II) *$\mathbb{C}_n$ has instances with exponentially sized minimal solutions.*

The proof of the exponential lower bound is by the following example. Each agent A_i, $2 \leq i \leq n$, has the strategy graph as follows:



The strategy graph for A_1 looks the same, except that its edges are unlabeled. This problem instance has a unique minimal coordinated plan of total length $2^{n+1} - 2$ transitions.

By showing that $\mathbb{C}_n$ is provably intractable, theorem 2 emphasizes our motivation for exploiting various structural restrictions on STS in order to find problem classes which are polynomial, e.g., $\mathbb{C}_k$, or at least belong to NP. In particular, the subclass of $\mathbb{C}_n$ strategy graphs of which are acyclic can be easily shown to belong to NP. However, the question of its exact hardness is still an open question. The same question is open for the decision version of $\mathbb{C}_n$.

Theorem 3 summarizes the complexity results for some other classes of STS (for notations see Section 4).

Theorem 3. **(I)** *There exists a polynomial time algorithm for $\mathbb{F}_n^\vee$ that determines existence of a coordinated plan and finds such a plan, if exists.* **(II)** *There exists a polynomial time algorithm for $\mathbb{C}_2^\pi$ that determines existence of a coordinated plan and finds such a plan, if exists.* **(III)** *$\mathbb{F}_n^\wedge$ is* NP*-complete.* **(IV)** *$\mathbb{F}_n^{\vee^\pi}$ is* NP*-complete.*

Proof sketch. *(I)* In $\mathbb{F}_n^\vee$, a single agent A_1 depends on a group of independent supporting agents $A_2, \ldots, A_n$. The algorithm is similar to that for $\mathbb{C}_2$. The idea is that several preconditions on the same transition can be checked independently. *(II)* $\mathbb{C}_2^\pi$ is an extension of $\mathbb{C}_2$ in which only *some* of transitions of the supported agent depend on the state of the supporting agent. The algorithm for $\mathbb{C}_2^\pi$ is similar to that for $\mathbb{C}_2$, except that a certain modification of the permitted-edge graph $G^{X|Y}$ is used. *(III),(IV)* In $\mathbb{F}_n^\wedge$, a

Main
Repeat *Pruning* while strategy sets are changing.
If all strategy sets are empty, then report that no coordinated plan exists.
Otherwise, perform *Construction*.
Pruning
For all edges (A_j, A_i) in $\mathcal{G}$ do:
For each $\sigma^i \in \mathcal{S}^i$ do:
If *Compatibility-Check*(σ_i, σ_j) returns false for all $\sigma^j \in \mathcal{S}^j$ then remove σ^i from $\mathcal{S}^i$.
For each $\sigma^j \in \mathcal{S}^j$ do:
If *Compatibility-Check*(σ_i, σ_j) returns false for all $\sigma^i \in \mathcal{S}^i$ then remove σ^j from $\mathcal{S}^j$.
Compatibility-Check (σ_i, σ_j)
If $l(\sigma^i)[j]$ with neighboring repetitions removed is a subsequence of σ^j,
then return true, else return false.
Construction
Pick any agent A and any strategy for it.
Traverse $\mathcal{G}$ undirectly from the node A. For each visited node A', choose any strategy of A'
that is compatible with the strategy chosen for the node from which we came to A'.

Fig. 6. Algorithm for $\mathbb{P}^{\pi}_{n}$.

single agent A_1 supports a group of agents $A_2, \ldots, A_n$. The membership in NP can be easily shown for both $\mathbb{F}^{\wedge}_{n}$ and $\mathbb{F}^{\vee^{\pi}}_{n}$. In turn, the proofs of hardness for $\mathbb{F}^{\wedge}_{n}$ and $\mathbb{F}^{\vee^{\pi}}_{n}$ are by polynomial reductions from 3-SAT and PATH WITH FORBIDDEN PAIRS problems, respectively.

Remarks: The algorithm for $\mathbb{C}_n$ can be extended for $\mathbb{C}^{\pi}_{n}$, similarly to the extension of the $\mathbb{C}_2$ algorithm for $\mathbb{C}^{\pi}_{2}$. Results presented for the cases of chain and inverse fork dependence graphs can be generalized to the case of dependence graph being a tree directed to its root.

Coordinating Bounded Agents In this section we consider the class of problems for which the number and size of strategies for each agent is polynomial in the size of the problem domain. In what follows, such agents are referred as *bounded*. Let us denote by $\mathcal{S}^i$ the set of allowed strategies σ^i of A_i (i.e., source-to-target paths in G^i). We show that if a dependence graph forms a polytree then STS for bounded agents is polynomial. However, its extension to singly connected directed graphs is already NP-complete.

Consider $\mathbb{P}^{\pi}_{n}$. Notice that any polytree is an acyclic graph. Denote by $l(\sigma^i)$ the sequence of edge labels along σ^i, and by $l_j(\sigma^i)[j]$ the projection of $l(\sigma^i)$ to the nodes of G^j. Figure 6 presents our algorithm for $\mathbb{P}^{\pi}_{n}$.

Theorem 4. (I) *There exists a polynomial time algorithm for $\mathbb{P}^{\pi}_{n}$ with bounded agents that determines existence of a coordinated plan and finds such a plan, if exists.* **(II)** *$\mathbb{S}^{\pi}_{n}$ for bounded agents is* NP*-complete.*

Proof sketch. (I) According to the analysis in Section 3, the condition of *Compatibility-Check* confirms that the checked pair σ^i, σ^j is coordinated. Therefore, convergence of the pruning process ensures that for each agent A: (i) any one of its strategies has at least one supporting strategy of each predecessor of A in $\mathcal{G}$, and (ii) any one of its strategies supports at least one strategy for any successor of A in $\mathcal{G}$. Hence, if *Construction* has a

personal strategy to begin with, then it is surely successful. Polynomiality holds, since each execution of *Pruning* decreases the total number of personal strategies.

(II) Membership in NP for $\mathbb{S}_n^\pi$ is straightforward, and the hardness proof is by a polynomial reduction from 3-SAT.

5 Discussion and Conclusions

In this paper, we concerned coordination for a set of agents that work on independent goals in the same environment. We investigated a particular family of the problems of finding coordinated plans, where actions of an agent depend on the local states of certain other agents. First, we presented a graphical representation model for this family of problems. Then, we classified these problems mainly according to the structural properties of the model. In the main part of the paper, we analyzed the computational properties of various problem classes. We gave polynomial solutions for several classes, while for some other we proved their NP-completeness or even provable intractability. A number of unsolved problems remains for the further research. The suggested approach—to find a sufficiently formal description for some problem classes, and to analyze their computational properties—seems to be prospective in the multi-agent area. Following some additional observations with respect to results presented in this paper.

First, let each agent and its states be considered as a multi-valued variable and its values, respectively, and let the set of agents' transitions be considered as a set of operators over the above variables. In this context, our results concern complexity analysis in the area of classical planning over multi-valued variables [4, 19]. Specifically, our results address the planning problems over multi-valued variables and only unary (= single effect) operators.

Second, in our model, each agent is assumed to be assigned to a personal goal, which is independent of the personal goals of all other agents. However, a supporting agent may have no explicit personal goal, while supporting activities of some other agents may be its only destiny. This particular relaxation can be immediately added into the presented model, with no negative impact on the computational properties of the problems. The coordinated group of dependent agents can be viewed as collaborating towards the achievement of some global goal.

Third, in this work we address only groups of fully controlled entities. Therefore, in particular, our results do not concern state-transition settings where agent's strategy depends on uncontrolled environment, like that studied in [3, 5].

In the future, we plan to continue with analysis of various classes of STS. In addition, we want to examine other forms of dependence between the agents, and other forms of goal(s) definition for a group of agents. This issues will be examined in the context of their impact on the complexity of coordination. We also plan to address related optimization problems.

References

1. R. Alami, F. Ingrand, and S. Qutub. A Scheme for Coordinating Multi-robot Planning Activities and Plans Execution. In *Proceedings of the 13th European Conference on Artificial Intelligence – ECAI*, Brighton, UK, 1998.

2. R. Alami, F. Robert, F. Ingrand, and S. Suzuki. Multi-robot Cooperation through Incremental Plan-Merging. In *International Conference on Robotics and Automation*, 1995.
3. R.C. Arkin and T. Balch. Cooperative Multiagent Robotic Systems. In *Artificial Intelligence and Mobile Robots*. MIT Press, 1998.
4. C. Bäckström and B. Nebel. Complexity Results for SAS^+ Planning. *Computational Intelligence*, 11(4):625–655, 1995.
5. T. Balch, G. Boone, T. Collins, H. Forbes, D. MacKenzie, and J. Santamaria. Io, Ganymede and Callisto - a Multiagent Robot Trash-Collecting Team. *AI Magazine*, 16(2):39–53, 1995.
6. R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.
7. Y. Cao, A. S. Fukunaga, and A. B. Kahng. Cooperative Mobile Robotics: Antecedents and Directions. *Autonomous Robots*, 4, 1997.
8. Bradley J. Clement and Edmund H. Durfee. Top-Down Search for Coordinating the Hierarchical Plans of Multiple Agents. In *Proceedings of the Third International Conference on Autonomous Agents*, pages 252–259, 1999.
9. P. Cohen and H. Levesque. Confirmations and Joint Actions. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 951–957, 1991.
10. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press.
11. K. Decker and V. Lesser. Designing a Family of Coordination Algorithms. In M. Huhns and M.Singh, editors, *Readings in Agents*, pages 450–457. Morgan Kaufmann, 1998.
12. M. d'Inverno, M. Luck, and M. Wooldridge. Cooperation Structures. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 600–605, 1997.
13. C. Domshlak and Y. Dinitz. Multi-Agent Off-line Coordination: Structure and Complexity. Technical Report CS-01-04, Department of Computer Science, Ben-Gurion University, 2001.
14. G. Dudek, M.R.M. Jenkin, E. Milios, and D. Wilkes. A Taxonomy for Multi-Agent Robotics. *Autonomous Robots*, 3(4):375–397, 1996.
15. E. Ephrati and J. S. Rosenschein. Divide and Conquer in Multi-Agent Planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 375–380, 1994.
16. M. Georgeff. Communication and Interaction in Multi-Agent Planning. In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 125–129, 1983.
17. B. Grosz. Collaborative Systems. *AI Magazine*, 17(2):67–85, 1995.
18. B. Grosz and S. Kraus. Collaborative Plans for Complex Group Action. *Artificial Intelligence*, 86(2):269–357, 1996.
19. P. Jonsson and C. Bäckström. State-variable Planning under Structural Restrictions: Algorithms and Complexity. *Artificial Intelligence*, 100(1–2):125–176, 1998.
20. D. N. Kinny, M. Ljungberg, A. S. Rao, E. S. Sonenberg, G. Tidhar, and E. Werner. Planned Team Activity. In *Artificial Social Systems*, volume 830 of *LNCS*, pages 227–256. 1994.
21. J. McCarthy and P. Hayes. Some Phylosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 6, 1969.
22. Y. Moses and M. Tennenholtz. Multi-entity Models. *Machine Intelligence*, 14:63–88, 1995.
23. C. Le Pape. A Combination of Centralized and Distributed Methods for Multi-agent Planning and Scheduling. In *Proceedings of the IEEE International Conference on Robotics and Automation – ICRA*, pages 488–493, 1990.
24. O. Shehory and S. Kraus. Formation of Overlapping Coalitions for Precedence-ordered Task Execution among Autonomous Agents. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 330–337. AAAI Press / MIT Press, 1996.
25. R. C. Smith. The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solving. *IEEE Transactions on Computers*, 29(12), 1994.
26. S. Yuta and S. Premvuti. Coordinating Autonomous and Centralized Decision Making to Achieve Cooperative Behaviors between Multiple Mobile Robots. In *Proceedings of International Conference on Intelligent Robots and Systems*, pages 1566–1574, 1992.

Approximate Planning for Factored POMDPs

Zhengzhu Feng[1] and Eric A. Hansen[2]

[1] Computer Science Department, University of Massachusetts, Amherst MA 02062
fengzz@cs.umass.edu,
[2] Computer Science Department, Mississippi State University, Mississippi State MS 39762
hansen@cs.msstate.edu

Abstract. We describe an approximate dynamic programming algorithm for partially observable Markov decision processes represented in factored form. Two complementary forms of approximation are used to simplify a piecewise linear and convex value function, where each linear facet of the function is represented compactly by an algebraic decision diagram. In one form of approximation, the degree of state abstraction is increased by aggregating states with similar values. In the second form of approximation, the value function is simplified by removing linear facets that contribute marginally to value. We derive an error bound that applies to both forms of approximation. Experimental results show that this approach improves the performance of dynamic programming and extends the range of problems it can solve.

1 Introduction

Markov decision processes (MDPs) have been adopted as a framework for research in decision-theoretic planning [2]. An MDP models planning problems for which actions have an uncertain effect on the state, and sensory feedback compensates for uncertainty about the state. An MDP is said to be completely observable if sensory feedback provides perfect state information before each action. It is said to be partially observable if sensory feedback is noisy and provides partial and imperfect state information. Although a partially observable Markov decision process (POMDP) provides a more realistic model, it is much more difficult to solve.

Dynamic programming is the most common approach to solving MDPs. However, a drawback of classic dynamic programming algorithms is that they require explicit enumeration of a problem's state space. Because the state space grows exponentially with the number of state variables, these algorithms are prey to Bellman's "curse of dimensionality." To address this problem, researchers have developed algorithms that exploit a factored state representation to create an abstract state space in which planning problems can be solved more efficiently. This approach was first developed for completely observable MDPs, using decision trees to aggregate states with identical values [3]. Improved performance was subsequently achieved using algebraic decision diagrams (ADDs) in place of decision trees [9]. A factored representation has also been exploited in solving POMDPs using both decision trees [4] and ADDs [8].

Although this approach improves the efficiency with which many problems can be solved, it provides no benefit unless there are states with identical values that can be aggregated. Moreover, the degree of state abstraction that can be achieved in this way may be insufficient to make a problem tractable. Many large MDPs and even small POMDPs resist exact solution. Thus, there is a need for approximation to allow a tradeoff between optimality and

computation time. For completely observable MDPs, an approach to approximation that increases the degree of state abstraction by aggregating states with similar (rather than identical) values has been developed, using both decision trees [5] and ADDs [13]. This paper develops a related approach to approximation for the partially observed case.

Whereas the value function of a completely observable MDP can be represented by a single ADD, the value function of a POMDP is represented by a set of ADDs. This makes development of an approximation algorithm for factored POMDPs more complex. We describe two complimentary forms of approximation for POMDPs. The first is closely related to approximation in the completely observable case and involves simplifying an ADD by aggregating states of similar value. In the second form of approximation, the set of ADDs representing the value function is reduced in size, by removing ADDs that contribute only marginally to value. The latter technique is often used in practice, but has not been analyzed before in the literature. We clarify the relationship between these two forms of approximation, and derive a bound on their approximation error. Experimental results show that this approach to approximation can improve the rate of convergence of dynamic programming, as well as the range of problems it can solve.

2 Partially observable Markov decision processes

We assume a discrete-time POMDP with finite sets of states S, actions A, and observations O. The transition function $Pr(s'|s,a)$, observation function $Pr(o|s',a)$, and reward function $r(s,a)$ are defined in the usual way. We assume a discounted infinite-horizon optimality criterion with discount factor $\beta \in (0,1]$. A belief state, b, is a probability distribution over S maintained by Bayesian conditioning. As is well-known, it contains all information necessary for optimal action selection. This gives rise to the standard approach to solving POMDPs. The problem is recast as a completely observable MDP with a continuous, $|S|$-dimensional state space consisting of all possible belief states, denoted $\mathcal{B}$. In this form, the POMDP can be solved by iteration of a *dynamic programming operator* T that improves a value function $V : \mathcal{B} \rightarrow \Re$ by performing the following "one-step backup" for all belief states b.

$$V_n(b) = TV_{n-1}(b) = \max_{a \in \mathcal{A}} \left\{ r(b,a) + \beta \sum_{o \in \mathcal{O}} Pr(o|b,a) V_{n-1}(b_o^a) \right\}, \tag{1}$$

where $r(b,a) = \sum_{s \in S} b(s) r(s,a)$; b_o^a is the updated belief state after action a and observation o; and $Pr(o|b,a) = \sum_{s,s' \in S} Pr(o,s'|s,a) b(s)$ where $Pr(o,s'|s,a) = Pr(s'|s,a) Pr(o|s',a)$.

The theory of dynamic programming tells us that the optimal value function V^* is the unique solution of the equation system $V = TV$, and that $V^* = \lim_{n \rightarrow \infty} T_n V_0$, where T_n denotes n applications of operator T to any initial value function V_0. We note that T_n corresponds to the value iteration algorithm.

An important result of Smallwood and Sondik [11] is that the dynamic-programming operator T preserves the piecewise linearity and convexity of the value function. A piecewise linear and convex value function V can be represented by a finite set of $|S|$-dimensional vectors of real numbers, $\mathcal{V} = \{v^0, v^1, \ldots, v^k\}$, such that the value of each belief state b is defined as:

$$V(b) = \max_{0 \leq i \leq k} \sum_{s \in S} b(s) v^i(s).$$

Several algorithms for computing the dynamic-programming operator have been developed. The most efficient, called incremental pruning [6], relies on the fact that the updated value function V_n of Equation (1) can be defined as a combination of simpler value functions, as follows:

$$V_n(b) = \max_{a \in A} V_n^a(b)$$
$$V_n^a(b) = \sum_{o \in \mathcal{O}} V_n^{a,o}(b)$$
$$V_n^{a,o}(b) = \frac{r(b,a)}{|\mathcal{O}|} + \beta Pr(o|b,a) V_{n-1}(b_o^a)$$

Each of these value functions is also piecewise linear and convex, and can be represented by a unique minimum-size set of vectors, denoted $\mathcal{V}_n$, $\mathcal{V}_n^a$, and $\mathcal{V}_n^{a,o}$ respectively.

The *cross sum* of two sets of vectors, $\mathcal{U}$ and $\mathcal{W}$, is defined as $\mathcal{U} \oplus \mathcal{W} = \{u + w | u \in \mathcal{U}, w \in \mathcal{W}\}$. An operator that takes a set of vectors $\mathcal{U}$ and reduces it to its unique minimum form is denoted PRUNE($\mathcal{U}$). Using this notation, the minimum-size sets of vectors defined above can be computed as follows:

$$\mathcal{V}_n = PRUNE\left(\cup_{a \in A} \mathcal{V}_n^a\right)$$
$$\mathcal{V}_n^a = PRUNE\left(\oplus_{o \in \mathcal{O}} \mathcal{V}_n^{a,o}\right)$$
$$\mathcal{V}_n^{a,o} = PRUNE\left(\{v^{a,o,i} | v^i \in \mathcal{V}_{n-1}\}\right),$$

where $v^{a,o,i}$ is the vector defined by

$$v^{a,o,i}(s) = \frac{r(s,a)}{|\mathcal{O}|} + \beta \sum_{s' \in S} Pr(o, s'|s, a) v^i(s'). \tag{2}$$

Incremental pruning gains its efficiency (and its name) from the way it interleaves pruning and cross-sum to compute V_n^a, as follows:

$$\mathcal{V}_n^a = PRUNE(...PRUNE(PRUNE(\mathcal{V}_n^{a,o_1} \oplus \mathcal{V}_n^{a,o_2}) \oplus \mathcal{V}_n^{a,o_3}) \ldots \oplus \mathcal{V}_n^{a,o_k}).$$

PRUNE reduces a set of vectors to a unique, minimal-size set by removing "dominated" vectors, that is, vectors that can be removed without affecting the value of any belief state. There are two tests for dominated vectors. One test determines that vector w is dominated by some other vector $u \in \mathcal{V}$ when

$$w(s) \leq u(s), \forall s \in S. \tag{3}$$

Although this test can be performed efficiently, it cannot detect all dominated vectors. Therefore, it is supplemented by a second, less efficient test that determines that a vector w is dominated by a set of vectors $\mathcal{V}$ when the following linear program cannot be solved for a value of d that is greater than zero.

variables: d, $b(s)\ \forall s \in S$
maximize d
subject to the constraints
$\sum_{s \in S} b(s)(w(s) - u(s)) \geq d,\ \forall u \in \mathcal{V}$
$\sum_{s \in S} b(s) = 1$

In a single iteration of incremental pruning, many linear programs must be solved to detect all dominated vectors, and the use of linear programming to test for domination has been found to consume more than 95% of the running time of incremental pruning [6]. The number of variables in each linear program is determined by the size of the state space. The number of constraints in each linear program (as well as the number of linear programs that need to be solved) is determined by the size of the vector set being pruned. This reflects the two principal sources of complexity in solving POMDPs. One source of complexity is shared by completely observable MDPs: the size of the state space. The other source of complexity is unique to POMDPs: the number of linear functions needed to represent the piecewise linear and convex value function.

In this paper, we explore two forms of approximation that address these two sources of complexity. We use state abstraction to reduce the number of variables in each linear program. We use relaxed tests for dominance to reduce the number of constraints in each linear program, as well as the number of linear programs that need to be solved. Before describing our approximation algorithm, we describe the framework for state abstraction.

3 State abstraction and algebraic decision diagrams

We consider an approach to state abstraction for MDPs and POMDPs that exploits a factored representation of the problem. We assume the relevant properties of a domain are described by a finite set of Boolean state variables, $\mathcal{X} = \{X_1, \ldots, X_n\}$, and observations are described by a finite set of Boolean observation variables $\mathcal{Y} = \{Y_1, \ldots, Y_m\}$. We define an abstract state as a partial assignment of truth values to $\mathcal{X}$, corresponding to a set of possible states.

Algebraic decision diagrams Instead of using matrices and vectors to represent the POMDP model, we use a data structure called an *algebraic decision diagram* (ADD) that exploits state abstraction to represent the model more compactly. Decision diagrams are widely used in VLSI CAD to represent and evaluate large state space systems [1]. A binary decision diagram is a compact representation of a Boolean function, $\mathcal{B}^n \rightarrow \mathcal{B}$. An algebraic decision diagram (ADD) generalizes a binary decision diagram to represent real-valued functions, $\mathcal{B}^n \rightarrow \Re$. Operations such as sum, product, and expectation (corresponding to similar operations on matrices and vectors) can be performed on ADDs, and efficient packages for manipulating ADDs are available [12]. Hoey *et al.* [9] show how to use ADDs to represent and solve completely observable MDPs. Hansen and Feng [8] extend this approach to POMDPs, based on earlier work of Boutilier and Poole [4]. In the rest of this section, we summarize this approach to state abstraction for POMDPs.

Model We represent the state transition function for each action a using a two-slice dynamic belief network (DBN). The DBN has two sets of variables, one set $\mathcal{X} = \{X_1, ..., X_n\}$ refers to the state before taking action a, and the other set $\mathcal{X}' = \{X'_1, ..., X'_n\}$ refers to the state after.

For each post-action variable X'_i, the conditional probability function $P^a(X'_i|\mathcal{X})$ of the DBN is represented compactly using an ADD. It is convenient to construct a single ADD, $P^a(\mathcal{X}'|\mathcal{X})$, that represents in factored form the state transition function for all post-action variables. Hoey *et al.* [9] call this a *complete action diagram* and describe the steps required to construct it.

The observation model of a POMDP is represented in factored form, in a similar way. We use an ADD $P^a(Y_i|\mathcal{X}')$ to represent the probability that observation variable Y_i is true after action a is taken and the state variables change to $\mathcal{X}'$. Given an ADD, $P^a(Y_i|\mathcal{X}')$, for each observation variable Y_i, it is again convenient to construct a single ADD, $P^a(\mathcal{Y}|\mathcal{X}')$, that represents in factored form the observation function for all observation variables. We call this a *complete observation diagram*, and it is constructed in the same way as a complete action diagram.

Given a complete action diagram and a complete observation diagram, a single ADD, $P^{a,o}(\mathcal{X}'|\mathcal{X})$, representing the transition probabilities for all state variables after action a and observation o, is constructed as follows:

$$P^{a,o}(\mathcal{X}'|\mathcal{X}) = P^a(\mathcal{X}'|\mathcal{X})P^a(\mathcal{Y}|\mathcal{X}').$$

The ADD $P^{a,o}(\mathcal{X}'|\mathcal{X})$ represents the probabilities $Pr(o, s'|s, a)$, just as $P^a(\mathcal{X}'|\mathcal{X})$ represents the probabilities $Pr(s'|s, a)$ and $P^a(\mathcal{Y}|\mathcal{X}')$ represents the probabilities $Pr(o|s', a)$.

The reward function for each action a can also be represented compactly by an ADD, denoted $R^a(\mathcal{X})$. Similarly, a piecewise linear and convex value function for a POMDP can be repesented compactly by a set of ADDs. We use the notation $\mathcal{V} = \{v^1(\mathcal{X}), v^2(\mathcal{X}), \ldots, v^k(\mathcal{X})\}$ to denote this value function.

Dynamic programming Hansen and Feng [8] describe how to modify the incremental pruning algorithm to exploit this factored representation of a POMDP for computational speedup. We briefly review their approach, and refer to their paper for details.

The first step of incremental pruning is generation of the linear functions in the sets $\mathcal{V}_n^{a,o}$. For a POMDP represented in factored form, the following equation replaces Equation (2):

$$v^{a,o,i}(\mathcal{X}) = \frac{R^a(\mathcal{X})}{|\mathcal{O}|} + \beta \sum_{\mathcal{X}'} P^{a,o}(\mathcal{X}'|\mathcal{X})v^i(\mathcal{X}').$$

All terms in this equation are represented by ADDs. The symbol $\sum_{\mathcal{X}'}$ denotes an ADD operator called existential abstraction that sums over the values of the state variables in $P^{a,o}(\mathcal{X}'|\mathcal{X})v^i(\mathcal{X}')$, exploiting state abstraction to compute the expected value efficiently.

State abstraction is also exploited to perform pruning more efficiently. Recall that the value function is represented by a set of linear functions. Each linear function is represented compactly by an ADD that can map multiple states to the same value, corresponding to a leaf of the ADD. In this case, the leaf corresponds to an abstract state. Hansen and Feng [8] describe an algorithm that finds a partition of the state space into abstract states that is consistent with the set of ADDs. Given this abstract state space, both tests for dominance can be performed more efficiently. In particular, the number of variables in the linear program used to test for dominace is reduced in proportion to the reduction in size of the state space. Because linear programming consumes most of the running time of incremental pruning, this significantly improves the performance of the algorithm in the best case. In the worst case, performance is only slightly worse since the overhead for this approach is almost negligible. Hansen and Feng [8] report speedups of up to a factor of twenty for the problems they test. The degree of speedup is proportional to the degree of state abstraction.

4 Approximation algorithm

As an approach to scaling up dynamic programming for POMDPs, the exact algorithm reviewed in the previous section has two limitations. First, there may not be sufficiently many (or even any) states with identical values to create an abstract state space that is small enough to be tractable. Second, although the size of the state space contributes to the complexity of POMDPs, the primary source of complexity is the potential exponential growth in the number of linear functions (ADDs) needed to represent the value function.

We now describe an approximate dynamic programming algorithm that addresses both of these limitations by ignoring differences of value less than some error threshold δ. It runs more efficiently than the exact algorithm because it computes a simpler, approximate value function for which we can bound the approximation error. There are two places in which the algorithm ignores value differences – in representing state values, and in representing values of belief states. These correspond to two complementary forms of approximation, one in which an ADD representing state values is simplified, and the other in which a set of ADDs representing belief state values is reduced in size. In other words, one form of approximation reduces the size of the state space (using state abstraction) and the other reduces the size of the value function (by pruning more aggressively). Before describing the algorithm, we define what we mean by approximate dynamic programming and present some theoretical results that allow us to bound the approximation error

Approximation with bounded error We begin by defining what we mean by an approximate value function and an approximate dynamic programming operator.

Definition 1. *A value function $\hat{V}$ approximates a value function V with approximation error δ if $\|V - \hat{V}\| \leq \delta$.* (Note that $\|V - \hat{V}\|$ denotes $\max_{b \in \mathcal{B}} |V(b) - \hat{V}(b)|$.)

Definition 2. *An operator $\hat{T}$ approximates the dynamic programming operator T if for any value function V, $\|TV - \hat{T}V\| \leq \delta$.*

We define an approximate value iteration algorithm $\hat{T}_n$ in the same way that we defined the value iteration algorithm T_n. The error between the approximate and exact n-step value functions is bounded as follows.

Theorem 1. *For any $n > 0$ and any value function V,*

$$\|T_n V - \hat{T}_n V\| \leq \frac{\delta}{1-\beta}.$$

To compute a bound on the error between a n-step approximate value function and the optimal value function, we use the Bellman residual between the $(n-1)$th and nth approximate value functions. The following theorem is essentially the same as Theorem 12.2.5 of Ortega and Rheinboldt [10], who studied approximate contraction mappings for systems of nonlinear equations, and Theorem 4.2 of Cheng [7], who first applied their result to POMDPs.

Theorem 2. *The error between the current and optimal value function is bounded as follows,*

$$\|\hat{T}_n V - V^*\| \leq \frac{\beta}{1-\beta} \|\hat{T}_n V - \hat{T}_{n-1} V\| + \frac{\delta}{1-\beta}.$$

Value iteration using an approximate dynamic programming operator converges "weakly," that is, two successive value functions fall within a distance $\frac{2\delta}{1-\beta}$ in the limit. (Decreasing δ after "weak" convergence will allow further improvement, as discussed later.)

Theorem 3. *For any value function V and $\varepsilon > 0$, there is an N such that for all $n > N$,*

$$\|\hat{T}_n V - \hat{T}_{n-1} V\| \leq \frac{2\delta}{1-\beta} + \varepsilon.$$

Simplifying ADDs We first describe an approach to approximation that simplifies an ADD by ignoring small differences in state values. It is based on a similar approach to approximation for completely observable MDPs [13], although modifications are needed to extend this approach to POMDPs.

For completely observable MDPs, a single ADD represents the value function. Each leaf of the ADD corresponds to a distinct value. If more than one state has the same value, the states are mapped to the same leaf. In this way, a leaf can represent a set of states, or equivalently, an abstract state. Because state abstraction can be exploited to accelerate dynamic programming, the approach to approximation is to increase the degree of state abstraction by aggregating states with similar (though not identical) values.

St-Aubin *et al.* [13] introduce the following notation and terminology. The value of a state is represented as a pair $[l, u]$, where the lower, l, and upper, u, bounds on the values are both represented. The *span* of a state, s, is given by $span(s) = u - l$. The *combined span* of states $s_1, s_2, \ldots, s_n$ with values $[l_1, u_1], \ldots, [l_n, u_n]$, is given by $cspan(s_1, s_2, \ldots, s_n) = max(u_1, \ldots, u_n) - min(l_1, \ldots, l_n)$. The method of approximation is to merge states (and correspondingly, leaves of an ADD) when their combined span is less than δ. This approximation is performed after each iteration of dynamic programming. The simpler ADD allows computational speedup, at the cost of some approximation error introduced by ignoring differences of value less than δ.

To implement this approach to approximation, St-Aubin *et al.* [13] modified the ADD package so that a leaf of an ADD can represent a range of values, and a single ADD can represent a *ranged value function*. We don't do this because the value function of a POMDP is represented by a set of ADDs, instead of a single ADD, and we are concerned with upper and lower bounds on the values of belief states. The set of ADDs representing the lower bound function may not be the same as the set of ADDs representing the upper bound function. An alternative to a ranged value function is to use two ADDs to represent bounds on state values – one for lower bounds and one for upper bounds. But this representation would require performing incremental pruning twice – once to compute a piecewise linear and convex lower bound function and once to compute a piecewise linear and convex upper bound function – doubling the complexity of the algorithm. Instead, we found that we can achieve an equally good result by computing a piecewise linear and convex lower bound function only, and representing the upper bound by using a scalar for the approximation error.

Figure 1 illustrates the effect of simplification. The ADD on the left is simplified by merging leaves that have a combined span of less than $\delta = 0.5$. The ADD on the right represents a lower bound on the value of each abstract state. Adding δ to each lower bound gives the upper bound. We use the following algorithm to simplify an ADD. The input is an ADD and approximation threshold δ. The output is a simplified ADD with bounded error.

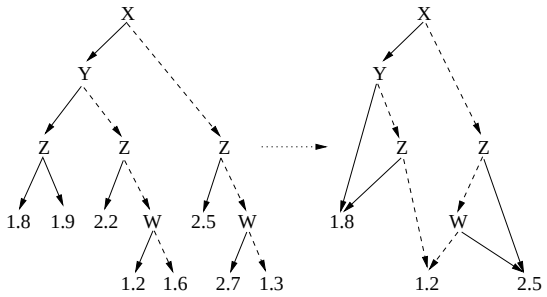


Fig. 1: Example of ADD simplification.

```
QUEUE ← all leaves of ADD sorted in increasing
            order of value
s ← remove first element from QUEUE
X ← {s}
while QUEUE is not empty
  t ← remove next element from QUEUE
  if cspan(X∪{t}) ≤ δ
    X ← X ∪ {t}
  else
    merge(X) and create new ADD leaf for X
    X ← {t}
  endif
endwhile
```

Because the complexity of each merge is $|S|$, the complexity of this algorithm is $O(|S|^2$. Performing this simplification algorithm on each ADD in a piecewise linear and convex value function results in an approximate value function with approximation error δ.

Theorem 4. *Let $V = \{v_1, \ldots, v_n\}$ be a piecewise linear and convex value function and let $V' = \{v'_1, \ldots, v'_n\}$ be its approximation such that for each v'_i, we have $||v_i - v'_i|| \leq \delta$. Then $||V - V'|| \leq \delta$.*

Pruning ADDs In Section 2, we described two tests for dominated linear functions. It has long been recognized that both tests are sensitive to numerical imprecision errors when the value representing the degree of dominace is close to zero. Thus, a precision parameter is typically used to prevent linear functions from being included in the value function due to numerical imprecision error. Instead of testing for a value greater than zero to ensure that a linear function is not dominated, the test is for a value greater that 10^{-15}, for example, or some number that represent the limit of numerical precision on the computer.

Many practitioners have noticed that increasing this precision parameter (to a value of, say, 10^{-5}), has the added benefit of pruning vectors that contribute only marginally to the value function. This often results in significant performance improvement. Although this technique is widely used in practice, the effect of this approximation on the error bound of the value function has not been analyzed before in the literature. We consider this second method

of approximation in this paper because of its close, and complementary, relationship to our first method of approximation, which also ignores small differences of value. Equation (3) gives a test for dominance that we generalize to allow approximation as follows.

Definition 3. *A linear function w is approximately dominated by another linear function $u \in \mathcal{V}$ when*

$$w(s) - \delta \leq u(s), \ \forall s \in S,$$

where $\delta > 0$.

The linear programming test for domination is generalized to allow approximation as follows.

Definition 4. *A linear function w is approximately dominated by a set of linear functions $\mathcal{V}$, if the output of the linear program, d, is less than $\delta > 0$.*

Let $PRUNE'$ be the pruning operator that employs these two approximate dominance tests.

Theorem 5. *For any set of vectors $\mathcal{V}$,*

$$\|PRUNE(\mathcal{V}) - PRUNE'(\mathcal{V})\| \leq \delta.$$

Accumulation of error Both the approximate ADD simplification algorithm and the approximate pruning algorithm are applied repeatedly during incremental pruning. They are applied to each set $\mathcal{V}_n^{a,o}$. They are applied to each of the $|O|$ sets of ADDs created by the cross-sum operator during the computation of $\mathcal{V}_n^a$. Finally, they are applied to the set $\mathcal{V}_n$ created by the union of the sets $\mathcal{V}_n^a$. Thus, we must consider how approximation error accumulates during the progress of incremental pruning.

Lemma 1. *If a set of ADDs representing value function V is simplified with approximation error δ^1 and then pruned with approximation error δ^2, the resulting set of ADDs represents a value function that approximates V with error $\delta^1 + \delta^2$.*

Lemma 2. *If $\hat{V^1}$ is an approximation of value function V^1 with approximation error δ^1, and $\hat{V^2}$ is an approximation of value function V^2 with approximation error δ^2, then:*

1. *$\hat{V^1} + \hat{V^2}$ is an approximation of $V^1 + V^2$ with approximation error $\delta^1 + \delta^2$, and*
2. *$\hat{V^1} \cup \hat{V^2}$ is an approximation of $V^1 \cup V^2$ with approximation error $\max(\delta^1, \delta^2)$.*

Theorem 6. *Let $\hat{T}$ denote an approximation of the dynamic programming operator T computed by incremental pruning with simplification error δ^1 and pruning error δ^2. For any value function V,*

$$\|TV - \hat{T}V\| \leq (2|\mathcal{O}| + 1)(\delta^1 + \delta^2).$$

Letting $\delta = (2|\mathcal{O}| + 1)(\delta^1 + \delta^2)$, we can use Theorem 2 to compute a bound on the error between the approximate and optimal value functions.

Adjustment of approximation Finally, we note that with exact dynamic programming, the difference between successive value functions always decreases from one iteration to the next. This is not necessarily the case with approximate dynamic programming. It suggests a strategy for reducing the approximation parameters over successive iterations. Whenever there is an increase in the Bellman residual, we reduce the approximation parameters (*e.g.*, by the discount factor 0.5) and the solution continues to improve. By using a high degree of approximation initially and gradually reducing it, we may accelerate the rate of improvement in initial iterations and still eventually achieve a result of equal quality as a result found by the exact algorithm. This is explored in the next section.

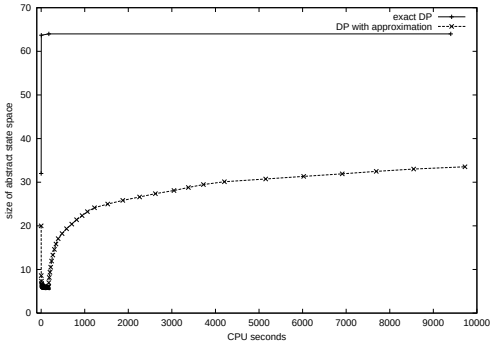


Fig. 2: Size of abstract state space using exact and approximate ADD simplification.

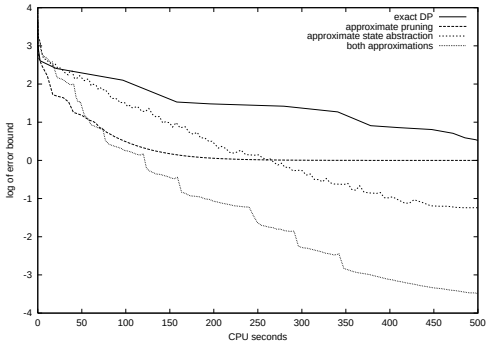


Fig. 3: Rate of convergence using both forms of approximation, separately and together.

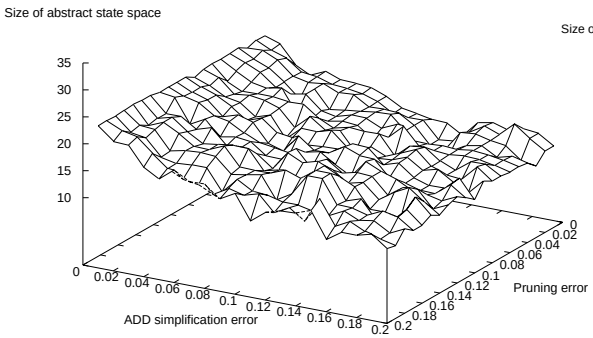


Fig. 4: Interaction between ADD simplification and pruning error on size of abstract state space.

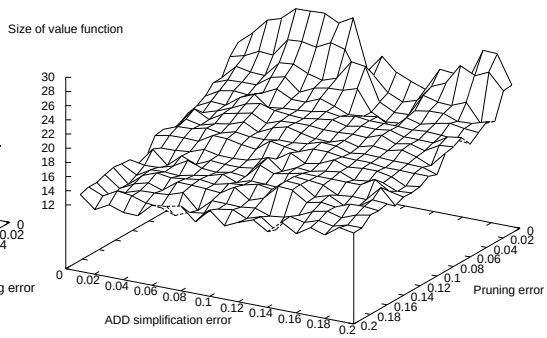


Fig. 5: Interaction between ADD simplification and pruning error on size of value function.

5 Analysis of performance

Hansen and Feng [8] use seven test problems to evaluate the performance of their exact dynamic programming algorithm for factored POMDPs. Among these, the fourth test problem (with six state variables, five actions, and two observation variables) illustrates the worst-case performance of the algorithm. Because there are strong dependencies among all the state variables, the algorithm finds no state abstraction. So we use this example as a test of whether the approximation algorithm can create state abstractions where the exact algorithm cannot. Figure 2 compares the size of the abstract state space created by the exact and approximation algorithms over successive iterations. Approximate ADD simplification makes it possible to solve the problem in an abstract state space that varies in size from five to thirty states, compared to 64 states in the original state space. This shows that ADD simplification can create useful state abstractions for problems with little or no variable independence.

Figure 4 and 5 show the interaction between the two forms of approximation by way of their effect on the two principal sources of POMDP complexity – the size of the state space and the size of the value function. The data is collected by running the program

with different pruning and simplification errors for 30 iterations. The average size of the abstract state space and the average size of the value function are then plotted as a function of the two types of approximation error. Figure 4 shows that increasing the ADD pruning error has little or no effect on ADD simplification. Figure 5 shows that increasing the ADD simplification error only slightly amplifies the effect of ADD pruning. Thus, the performance improvement achieved from each form of approximation is almost independent, with a slight positive interaction effect. The two methods of approximation are complementary. ADD simplification decreases the size of the state space and ADD pruning decreases the size of the value function.

Figure 3 shows the separate effect of each form of approximation on the rate of convergence, as well as their combined effect. (The ADD simplification error is 0.1 and the ADD pruning error is 0.01.) It shows that using both forms of approximation results in better performance than using either one alone. It also shows that the approximation algorithm can find a better solution than the exact algorithm, in the same amount of time. The reason for this is that the approximation algorithm approximates the dynamic-programming operator, which performs a single iteration of dynamic programming. Dynamic programming takes many iterations to converge. Because approximation allows the dynamic-programming operator to be computed faster in exchange for slightly less improvement of the value function, approximation can have the effect of increasing the *rate* of improvement. In other words, the approximation algorithm can perform more iterations in the same amount of time, and, as a result, can find a better solution in the same amount of time. This is true even though each iteration of the approximation algorithm may not improve the value function as much as a corresponding iteration of the exact algorithm.

We are not only interested in improving the rate of convergence of dynamic programming. We are also interested in solving larger problems than the exact algorithm can solve. The scalability of both the exact algorithm and the approximation algorithm is limited by the same two factors - the size of the state space and the size of the value function. (The dynamic programming algorithm currently cannot handle problems with more than about 50 states or value functions with more than a few hundred ADDs.) The approximation algorithm scales better than the exact algorithm because it can control the size of the state space and the size of the value function. It controls the size of the (abstract) state space by using approximation to adjust the degree of state abstraction. It controls the size of the value function by using approximation to adjust the threshold for pruning ADDs, and thus the number of ADDs that are pruned. This allows the algorithm to find approximate solutions to problems with more states than the exact algorithm can handle, and to avoid an exponential explosion in the size of the value function. The quality of the solution that can be found within these limitations is problem-dependent, but easily estimated by computing the error bound.

6 Conclusion

POMDPs are very difficult to solve exactly and it is widely-recognized that approximation is needed to solve realistic problems. We have described two complementary forms of approximation that improve the performance of a dynamic programming algorithm that computes a piecewise linear and convex value function. The first form of approximation increases the degree of state abstraction by ignoring state distinctions that have little effect on value. The second form of approximation reduces the number of linear functions used to represent the

value function by removing those that have little effect on value. Both forms of approximation allow computational speedup in exchange for a bounded decrease in solution quality. Both also have tunable parameters that allow the degree of approximation to be adjusted to suit the problem. We showed that this approach to approximation improves both the rate of convergence of dynamic programming, and its scalability.

References

1. Bahar, R.I.; Frohm, E.A.; Gaona, C.M.; Hachtel, G.D.; Macii, E.; Pardo, A.; and Somenzi, F. Algebraic decision diagrams and their applications. International Conference on Computer-Aided Design, 188–191, IEEE, 1993.
2. Boutilier, C.; Dean, T.; and Hanks, S. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94, 1999.
3. Boutilier, C.; Dearden, R.; and Goldszmidt, M. Exploiting structure in policy construction. In Proceedings of the Fourteenth International Conference on Artificial Intelligence (IJCAI-95), 1104–1111, Montreal, Canada, 1995.
4. Boutilier, C. and Poole, D. Computing optimal policies for partially observable decision processes using compact representations. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), 1168–1175, Portland, OR, 1996.
5. Boutilier, C. and Dearden, R. Approximating value trees in structured dynamic programming. In Proceedings of the Fourteenth International Conference on Machine Learning, 54–62. Bari, Italy, 1996.
6. Cassandra, A.R.; Littman, M.L.; and Zhang, N.L. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97), 54–61, Providence, RI, 1997.
7. Cheng, H. *Algorithms for Partially Observable Markov Decision Processes.* PhD Thesis, University of British Columbia, 1988.
8. Hansen, E. and Feng, Z. Dynamic programming for POMDPs using a factored state representation. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling, 130-139, Menlo Park, CA: AAAI Press, 2000.
9. Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. SPUDD: Stochastic Planning using Decision Diagrams. In Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99), Stockholm, Sweden, 1999.
10. Ortega, J.M and Rheinboldt, W.C. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press: New York, 1970.
11. Smallwood, R.D. and Sondik, E.J. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research* 21:1071–1088, 1973.
12. Somenzi, F. CUDD: CU decision diagram package. Available from ftp://vlsi.colorado.edu/pub/, 1998.
13. St-Aubin, R.; Hoey, J.; and Boutilier, C. APRICODD: Approximate Policy Construction using Decision Diagrams. In Advances in Neural Information Processing Systems 13 (NIPS-00): Proceedings of the 2000 Conference, Denver, CO, 2000.

Solving Informative Partially Observable Markov Decision Processes

Weihong Zhang and **Nevin L. Zhang**

Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong, China

Abstract. Solving Partially Observable Markov Decision Processes (POMDPs) generally is computationally intractable. In this paper, we study a special POMDP class, namely informative POMDPs, where each observation provides good albeit incomplete information about world states. We propose two ways to accelerate value iteration algorithm for such POMDPs. First, dynamic programming (DP) updates can be carried out over a relatively small subset of belief space. Conducting DP updates over subspace leads to two advantages: representational savings in space and computational savings in time. Second, a point-based procedure is used to cut down the number of iterations for value iteration over subspace to converge. Empirical studies are presented to demonstrate various computational gains.

1 Introduction

Partially Observable Markov Decision Processes (POMDPs) provide a general framework for AI planning problems where effects of actions are nondeterministic and the state of the world is not known with certainty. Unfortunately, solving general POMDPs is computationally intractable [10]. For this reason, special classes of POMDPs incur much attention recently in the community(e.g., [8, 12]).

In this paper, we study a class of POMDPs, namely *informative POMDPs*, where any observation can restrict the world into a small set of states. Informative POMDPs come to be a median ground in terms of informative degree of observations. In one extreme case, unobservable POMDPs assume that observations do not provide any information about world states(e.g.,[9]). In other words, an observation cannot restrict the world into any range of states. In another extreme case, fully observable MDPs assume that an observation restricts the world into a unique state.

For informative POMDPs, we propose two ways to accelerate value iteration. First, for such POMDPs, we observe that dynamic programming(DP) updates can be carried out over a subset of belief space. DP updates over a subset leads to two advantages: fewer vectors are in need to represent a value function over a subset;computational savings are gained in computing sets of vectors representing value functions over the subset. Second, to further enhance our capability of solving informative POMDPs, a point-based procedure is integrated into value iteration over the subset [13]. The procedure effectively cuts down the number of iterations for value iteration to converge.

The integrated algorithm is able to solve an informative POMDP with 105 states, 35 observations and 5 actions within 430 CPU seconds.

The rest of the paper is organized as follows. In next section, we introduce background knowledge and conventional notations. In Section 3, we discuss problem characteristics of informative POMDPs and problem examples in the literature. In Section 4, we show how the problem characteristics can be exploited in value iteration. Section 5 reports experiments on comparing value iteration over belief space and over a subset of it. In Section 6, we integrate the point-based procedure to value iteration over a subset of belief space. In Section 7, we briefly discuss some related work.

2 Background

In a POMDP model, the environment is described by a set of states $\mathcal{S}$. The agent changes the states by executing one of a finite set of actions $\mathcal{A}$. At each point in time, the world is in one state s. Based on the information it has, the agent chooses and executes an action a. Consequently, it receives an *immediate reward* $r(s,a)$ and the world moves stochastically into another state s' according to a *transition probability* $P(s'|s,a)$. Thereafter, the agent receives an observation z from a finite set $\mathcal{Z}$ according to an *observation probability* $P(z|s',a)$. The process repeats itself.

Information that the agent has about the current state of the world can be summarized by a probability distribution over $\mathcal{S}$ [1]. The probability distribution is called a *belief state* and is denoted by b. The set of all possible belief states is called the *belief space* and is denoted by $\mathcal{B}$. A *belief subspace* or simply *subspace* is a subset of $\mathcal{B}$. If the agent observes z after taking action a in belief state b, its next belief state b' is updated as

$$b'(s') = kP(z|s',a)\sum_{s} P(s'|s,a)b(s) \tag{1}$$

where k is a re-normalization constant. We will sometimes denote this new belief state by $\tau(b,a,z)$.

A *policy* prescribes an action for each possible belief state. In other words, it is a mapping from $\mathcal{B}$ to $\mathcal{A}$. Associated with policy π is its *value function* V^π. For each belief state b, $V^\pi(b)$ is the expected total discounted reward that the agent receives by following the policy starting from b, i.e. $V^\pi(b) = E_{\pi,b}[\sum_{t=0}^{\infty}\lambda^t r_t]$, where r_t is the reward received at time t and λ ($0 \leq \lambda < 1$) is the *discount factor*. It is known that there exists a policy π^* such that $V^{\pi^*}(b) \geq V^\pi(b)$ for any other policy π and any belief state b. Such a policy is called an *optimal policy*. The value function of an optimal policy is called the *optimal value function*. We denote it by V^*. For any positive number ϵ, a policy π is *ϵ-optimal* if $V^\pi(b) + \epsilon \geq V^*(b)$ for any belief state b.

The *dynamic programming(DP) update operator* T maps a value function V to another value function TV that is defined as follows: for any b in $\mathcal{B}$,

$$TV(b) = \max_a [r(b,a) + \lambda \sum_z P(z|b,a)V(\tau(b,a,z))]$$

where $r(b,a) = \sum_s r(s,a)b(s)$ is the expected reward if action a is taken in b.

Value iteration is an algorithm for finding ϵ-optimal value functions. It starts with an initial value function V_0 and iterates using the formula: $V_n = TV_{n-1}$. Value iteration terminates when the *Bellman residual* $\max_b |V_n(b) - V_{n-1}(b)|$ falls below $\epsilon(1-\lambda)/2\lambda$. When it does, the value function V_n is ϵ-optimal.

Value function V_n is *piecewise linear and convex (PLC)* and can be represented by a finite set of $|\mathcal{S}|$-dimensional *vectors* [11]. It is usually denoted by $\mathcal{V}_n$. In value iteration, a DP update computes a set $\mathcal{V}_{n+1}$ representing V_{n+1} from $\mathcal{V}_n$ representing V_n.

3 Problem Characteristics

In general, a POMDP agent perceives the world by receiving observations. Starting from any state, if the agent executes an action a and receives an observation z, world states can be categorized into two classes by the observation model: states the agent can reach and states it cannot. Formally, the set of reachable states is $\{s|s \in \mathcal{S} \text{ and } P(z|s,a) > 0\}$. We denote it by $\mathcal{S}_{az}$.

An $[a,z]$ pair is said to be *informative* if the size $|\mathcal{S}_{az}|$ is much smaller than $|\mathcal{S}|$. Intuitively, if the pair $[a,z]$ is informative, after executing a and receiving z, the agent knows that the true world states are restricted into a small set. An observation z is said to be *informative* if $[a,z]$ is informative for every action a giving rise to z. Intuitively, an observation is informative if it always gives the agent an good idea about world states regardless of the action executed at previous time point. A POMDP is said to be *informative* if all observations are informative. In other words, any observation the agent receives always provides it a good idea about world states. Since one observation is received at each time point, a POMDP agent always has a good albeit imperfect idea about the world.

Informative POMDPs are especially suitable and appropriate for modeling a class of problems. In this class, a problem state is described by a number of variables(fluents). Some variables are observable while others are not. The possible assignments to observable variables form the observation space. A specific assignment to observable variables restricts the world states into a small range of them. A *slotted Aloha* protocol problem belongs to this class [2, 4]. Similar problem characteristics also exist in a non-stationary environment model proposed for reinforcement learning [7].

4 Exploiting Problem Characteristics

In this section, we show how informativeness can be exploited in value iteration. We start from belief subspace representation.

4.1 Belief subspace

We are interested in particular subspace type: *belief simplex*. It is specified by a list of *extreme belief states*. The simplex with extreme belief states b_1, b_2, ...,b_k consists of all belief states of the form $\sum_{i=1}^{k} \lambda_i b_i$ where $\lambda_i \geq 0$ and $\sum_{i=1}^{k} \lambda_i = 1$.

Suppose the current belief state is b. If the agent executes an action a and receives an observation z, its next belief state is $\tau(b,a,z)$. If we vary the belief state in the belief

space $\mathcal{B}$, we obtain a set $\{\tau(b,a,z)|b \in \mathcal{B}\}$. Abusing notation, we denote this set by $\tau(\mathcal{B},a,z)$. In words, no matter which belief state the agent starts from, if it receives z after performing a, its next belief state must be in $\tau(\mathcal{B},a,z)$. Obviously, $\tau(\mathcal{B},a,z) \subseteq \mathcal{B}$.

Belief states in the set $\tau(\mathcal{B},a,z)$ have nice property which can be explored in context of informative POMDPs. By belief state update equation, if state s' is not in the set $\mathcal{S}_{az}$, the belief $b'(s')$ equals 0. The nonzero beliefs must distribute over states in $\mathcal{S}_{az}$. To reveal the relation between belief states and the set $\mathcal{S}_{az}$, we define a subset of $\mathcal{B}$:

$$\phi(\mathcal{B},a,z) = \{b| \sum_{s\in\mathcal{S}_{az}} b(s) = 1.0,\ \forall s \in \mathcal{S}_{az}, b(s) \geq 0\}.$$

It can be proven that for any belief state b, $\tau(b,a,z)$ must be in the above set. Therefore, $\tau(\mathcal{B},a,z)$ is a subset of $\phi(\mathcal{B},a,z)$ for a pair $[a,z]$. It is easy to see that $\phi(\mathcal{B},a,z)$ is a simplex in which each extreme point has probability mass on one state.

We consider the union of subspaces $\cup_{a,z}\phi(\mathcal{B},a,z)$ for all possible combinations of actions and observations. It consists of all the belief states the agent can encounter. In other words, the agent can never get out of this set. To ease presentation, we denote this set by $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$. Since each simplex in it is a subset of $\mathcal{B}$, so is $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$.

One example on belief space and subspaces is shown in Figure 1. A POMDP has four states and four observations. Its belief region is the tetrahedron ABCD where A, B, C and D are extreme belief states. For simplicity, we also use these letters to refer to the states. Suppose that $\mathcal{S}_{az}$ sets are independent of the actions. More specifically, for any action a, $\mathcal{S}_{az_0} = \{A,B,C\}$, $\mathcal{S}_{az_1} = \{A,B,D\}$, $\mathcal{S}_{az_2} = \{A,C,D\}$, and $\mathcal{S}_{az_3} = \{B,C,D\}$. In this POMDP, belief simplexes are four facets ABC, ABD, ACD and BCD and belief subspace $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$ is the surface of the tetrahedron. We also note that the subspace $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$ is much smaller than $\mathcal{B}$ in size.

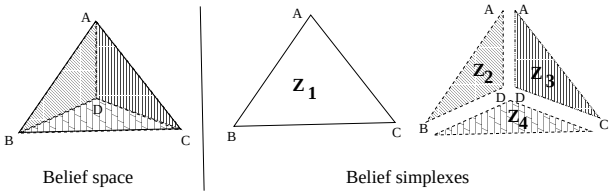


Fig. 1. Belief space, belief simplexes and belief subspace

4.2 Value functions over subspaces

A value function V_n over belief space $\mathcal{B}$ is a mapping from the belief space $\mathcal{B}$ to real line. Conceptually, for any b in $\mathcal{B}$, $V_n(b)$ is the maximum rewards the agent can receive in n steps if it starts from b. *Value function over subspace* is defined similarly. A n-step *value function over simplex* $\phi(\mathcal{B},a,z)$ is a mapping from the simplex. We denote it by $V_n^{\phi(\mathcal{B},a,z)}$. Conceptually, for a belief state b in subspace $\phi(\mathcal{B},a,z)$, $V_n^{\phi(\mathcal{B},a,z)}(b)$ is the

maximum rewards the agent can receive if it starts from b. An n-step value function $V_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}(b)$ can be defined similarly and its domain is restricted to $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$.

Value function V_n can be represented by a set $\mathcal{V}_n$ of $|\mathcal{S}|$-dimensional vectors. If V_n is restricted to a simplex $\phi(\mathcal{B},a,z)$, it is a value function $V_n^{\phi(\mathcal{B},a,z)}$ over the simplex. It preserves the PLC property and can be represented by a set of vectors. For informative POMDPs, the restriction will result in a representational advantage. Specifically, for a pair $[a,z]$, since the beliefs over states outside $\mathcal{S}_{az}$ are zero, we need to allocate only $|\mathcal{S}_{az}|$ components for a vector. Typically, a value function is represented by a great number of vectors. If one represents the same value function over a simplex, it would lead to tremendous savings because the vectors are of smaller dimensions.

Given a collection $\{\mathcal{V}_n^{\phi(\mathcal{B},a,z)}\}$ in which each set $\mathcal{V}_n^{\phi(\mathcal{B},a,z)}$ is associated with an underlying set $\mathcal{S}_{az}$, defining a value function $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ over subspace $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$ exhibits a little bit difficulty. This is because the underlying set $\mathcal{S}_{az}$ contains different states for different $[a,z]$ pairs. It makes no sense if one defines the value function by computing the inner product of a vector and a belief state because possibly the dimension of the vector differs from that of the belief state. We define $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ this way: for any b in $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$,

$$\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}(b) = \mathcal{V}_n^{\phi(\mathcal{B},a,z)}(b) \tag{2}$$

where $[a,z]$ is a pair such that $b \in \phi(\mathcal{B},a,z)$. The set $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ can be regarded as a two-dimensional array of sets over simplexes. When it needs to determine a value for a belief state, one (1) identifies a simplex containing it and (2) computes the value using the corresponding set of vectors. Obviously, the set $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ represents value function $V_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$.

4.3 DP update over subspace

In this subsection, we show how to conduct implicit DP update over belief subspace. The problem is cast as: given an array $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ representing $V_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ over subspace $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$, how to compute an array $\mathcal{V}_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$?

To compute the set $\mathcal{V}_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$, we construct one set $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a',z')}$ for any possible pair $[a',z']$. Before doing so, we recall how DP update over belief space constructs a vector in set $T\mathcal{V}_n$.

DP update $T\mathcal{V}_n$ computes a set $\mathcal{V}_{n+1}$ from a current set $\mathcal{V}_n$. It is known that each vector in $\mathcal{V}_{n+1}$ can be defined by a pair of action and a mapping from the set of observations to the set $\mathcal{V}_n$. Let us denote the action by a and the mapping by δ. For an observation z, we use δ_z to denote the mapped vector in $\mathcal{V}_n$. Given an action a and a mapping δ, the vector, denoted by $\beta_{a,\delta}$, is defined as follows: for each s in $\mathcal{S}$,

$$\beta_{a,\delta}(s) = r(s,a) + \lambda \sum_z \sum_{s'} P(s'|s,a)P(z|s',a)\delta_z(s').$$

By enumerating all possible combinations of actions and mappings, one can define different vectors. All these vectors form a set $\mathcal{V}_{n+1}$, i.e., $\{\beta_{a,\delta}|a \in \mathcal{A}, \delta : \mathcal{Z} \to \mathcal{V}_n\}$. It turns out that this set represents value function V_{n+1}.

We move forward to define a vector in $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a',z')}$ given an array $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$. Similar to the case in DP update $T\mathcal{V}_n$, a vector in set $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a',z')}$ can be defined by a pair of action a and a mapping δ but with two important modifications. First, the mapping δ is from set of observations to the array $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$. Moreover, for an observation z, δ_z is a vector in $\mathcal{V}_n^{\phi(\mathcal{B},a,z)}$. Second, the vector only need to be defined over the set $\mathcal{S}_{a'z'}$. To be precise, given a pair $[a', z']$, an action a and a mapping δ, a vector, denoted by $\beta_{a,\delta}$, can be defined as follows:

for each s in $\mathcal{S}_{a'z'}$,

$$\beta_{a,\delta}(s) = r(s,a) + \lambda \sum_z \sum_{s' \in \mathcal{S}_{az}} P(s'|s,a)P(z|s',a)\delta_z(s').$$

A couple of remarks are in order for the above definition. First, $\beta_{a,\delta}$ has only $|\mathcal{S}_{a'z'}|$ components. For states outside $\mathcal{S}_{a'z'}$, it is unnecessary to allocate space for them. Second, given the action a and observation z, when we define the component $\beta_{a,\delta}(s)$, we only need to account for next states in $\mathcal{S}_{az}$. This is true because for other states the probabilities of observing z are zero. It is important to note that an $|\mathcal{S}_{a'z'}|$-dimensional vector $\beta_{a,\delta}$ is constructed by making use of $|\mathcal{Z}|$ vectors: these vectors are of different dimensions because they come from different representing sets over simplexes.

If we enumerate all possible combinations of actions and mappings above, we can define various vectors. These vectors form a set

$$\{\beta_{a,\delta} | a \in \mathcal{A}, \delta : \mathcal{Z} \to \mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})} \ \& \ \forall z, \delta_z \in \mathcal{V}_n^{\phi(\mathcal{B},a,z)}\}.$$

The set is denoted by $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a',z')}$. The following lemma reveals the relation between the set and value function $V_{n+1}^{\phi(\mathcal{B},a',z')}$.

Lemma 1. *For any pair $[a, z]$, the set $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a,z)}$ represents value function $V_{n+1}^{\phi(\mathcal{B},a,z)}$ over simplex $\phi(\mathcal{B}, a, z)$.* □

For now, we are able to construct a set $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a,z)}$ for a pair $[a, z]$. A complete DP update over $\phi(\mathcal{B}, \mathcal{A}, \mathcal{Z})$ needs to construct such sets for all possible pairs of actions and observations. After these sets are constructed, they are pooled together to form an array $\mathcal{V}_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$. It induces a value function by (2). It can be proved that the array $\mathcal{V}_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ represents value function $V_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ over the set $\phi(\mathcal{B}, \mathcal{A}, \mathcal{Z})$. The following theorem means that $V_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ defines the same value function as V_{n+1} over the set $\phi(\mathcal{B}, \mathcal{A}, \mathcal{Z})$.

Theorem 1. *For any b in $\phi(\mathcal{B}, \mathcal{A}, \mathcal{Z})$, $V_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}(b) = V_{n+1}(b)$.* □

As a corollary of the above theorem, we remark that, if b is a belief state in the intersection of two simplexes $\phi(\mathcal{B}, a_1, z_1)$ and $\phi(\mathcal{B}, a_2, z_2)$ for two pairs $[a_1, z_1]$ and $[a_2, z_2]$, $V_{n+1}^{\phi(\mathcal{B},a_1,z_1)}(b) = V_{n+1}^{\phi(\mathcal{B},a_2,z_2)}(b)$.

4.4 Complexity analysis

DP update $T\mathcal{V}_n$ improves values for belief space $\mathcal{B}$, while DP update of computing $\mathcal{V}_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ from $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ improves values for subspace $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$. Since the subspace is much smaller than $\mathcal{B}$ in an informative POMDP, one expects:(1)fewer vectors are in need to represent a value function over a subspace;(2)since keeping useful vectors needs solve linear programs, this would lead to computational gains in time cost. Our empirical studies confirmed these two expectations.

4.5 Value iteration over subspace

Value iteration over subspace starts with a value function $\mathcal{V}_0^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$. Each set in it is initialized to contain a zero-vector of $|\mathcal{S}_{az}|$-dimension.

As value iteration continues, the Bellman Residual becomes smaller between two consecutive value functions over $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$. When the residual over $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$ falls below a predetermined threshold, it is also the case for the residual over any simplex. This suggest that the stopping criterion depend on residuals over simplexes. When the quantity $\max_{a,z}\max_{b\in\phi(\mathcal{B},a,z)}|\mathcal{V}_{n+1}^{\phi(\mathcal{B},a,z)}(b)-\mathcal{V}_n^{\phi(\mathcal{B},a,z)}(b)|$, the maximal difference between two consecutive value functions over all simplexes, falls below a threshold η, value iteration should terminate.

When value iteration terminates, it outputs the array $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$. A value function V over the entire belief space can be defined by one step lookahead operator as follows:

$$V(b)=\max_a\{r(b,a)+\sum_z P(z|b,a)\mathcal{V}_n^{\phi(\mathcal{B},a,z)}(\tau(b,a,z))\}\ \ \forall\ b\in\mathcal{B}. \tag{3}$$

The value function V defined is said to be $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$*-greedy*.

The hope is that if $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ is a good value function over $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$, so is $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$-greedy value function. The following theorem shows how the threshold η impacts the quality of value function $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ and $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$-greedy value function.

Theorem 2. *If $\eta\le\epsilon(1-\lambda)/(2\lambda|\mathcal{Z}|)$ and value iteration over $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$ outputs $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$, then $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$-greedy value function is ϵ-optimal over the entire belief space.* □

This theorem is important for two reasons. First, although value iteration over subspace computes a value function over a subset of belief space, ϵ-optimal value function over the entire belief space can be obtained by one step lookahead operator. Second, due to the availability of ϵ-optimal value function over $\mathcal{B}$, the agent can use it to select action for any belief state in $\mathcal{B}$. This is true for any initial belief states. Although $\phi(\mathcal{B},\mathcal{A},\mathcal{Z})$ consists of all belief states the agent can encounter after receiving any observation, the initial belief state does not necessarily belong to this set. The theorem implies that the $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$-greedy value function can be used to guide the agent to select near optimal action for any initial belief state.

Finally, we note that to guarantee the ϵ-optimality, the threshold η (set to $\epsilon(1-\lambda)/(2\lambda|\mathcal{Z}|)$) in value iteration over subspace is smaller than that over belief space.

This stopping criterion is said to be *strict* one. If η is set to be $\epsilon(1-\lambda)/(2\lambda)$ for value iteration over subspace, the condition is the *loose stopping criterion*. In our experiments, we use the loose stopping criterion.

5 Experiments

Experiments have been designed to test the performances of value iteration algorithms with and without exploiting the informative characteristics. Here we report results on a 3x3 grid world problem in Figure 2. It has nine states and the location 8(marked by $*$) is the goal state. The grid is divided by three rows and three columns. There are three locations along any row or column. The agent can perform one of four nominal-direction moving actions or a declaring success action. After performing a moving action, the agent reaches a neighboring location with probability 0.80 and stays at the same location with probability 0.20. Reasonable constraints are imposed to moving actions. For instance, if the agent is in location 0 and moves north, it stays at the same location. A declare-success action does not change the agent's location. After performing any action, the agent is informed of the column number with certainty. As such, the problem has three observations: col-0, col-1 and col-2. A move action incurs a cost of -1. If the agent declares success in location 8, it receives a reward of 10;If it does so in other locations, it receives a cost of -2.

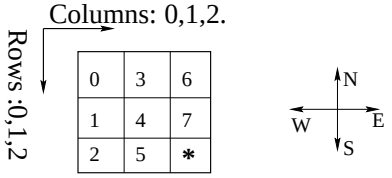


Fig. 2. A 3x3 grid world

This POMDP is informative. If value iteration is conducted without exploiting informativeness, one need to improve values over space $\mathcal{B}(=\{b|\sum_{i=0}^{8} b(s_i)=1.0\}$. Since the observations are column numbers and independent of actions, DP update over subspace need to account for three simplexes: $\mathcal{B}_j = \{b|\sum_{3j,3j+1,3j+2} b(s_j) = 1.0\}$ for j=0,1,2 where j is the column number.

Our experiments are conducted on a SUN SPARC workstation. The discount factor is set to 0.95. The precision parameter is set to 0.000001. The quality requirement ϵ is set to 0.01. We use the loose stopping criterion. In our experiments, incremental pruning [12, 5] is used to compute sets of vectors representing value functions over belief space or subspace. For convenience, we use `VI1` and `VI` to refer to the value iteration algorithms with and without exploiting regularities respectively. We compare `VI` and `VI1` at each iteration along two dimensions: the size of set representing value function and time cost to conduct a DP update. The results are presented in Figure 3.

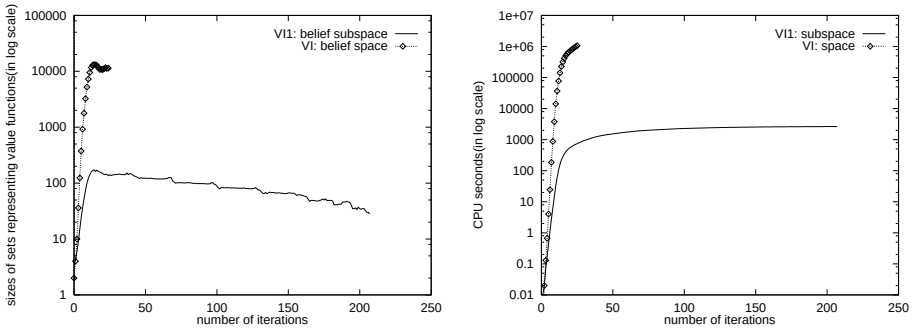


Fig. 3. Comparative study on value iterations over belief space and belief subspace

The first chart in the figure depicts the number of vectors in log-scale generated at each iteration for `VI` and `VI1`. In `VI`, at each iteration, we collects the sizes of sets representing value functions. In `VI1`, we compute three sets representing value functions over three simplexes and report the sum of the sizes of these three sets. For this problem, except the first iterations, `VI` generates significantly more vectors than `VI1`. In `VI`, after a severe growth, the number of vectors tends to be stable. In this case, value functions over belief space are represented by over 10,000 vectors. In contrary, the number of vectors generated by `VI1` is much smaller. Our experiments show that the maximum number is below 150. After `VI1` terminates, the value function is represented by only 28 vectors.

Due to the big difference between numbers of vectors generated by `VI1` and `VI`, `VI1` is significantly efficient than `VI`. This is demonstrated in the second chart in Figure 3. Note that CPU times in the figure are drawn in log-scale. When `VI1` terminates after 207 iterations, it takes around 2,700 seconds. On average, one DP update takes less than 13 seconds. For `VI`, it never terminates within reasonable time limit. By our data, it takes 1,052,590 seconds for first 25 iterations. On average, each iteration takes around 42,000 seconds. Comparing with `VI1`, we see that `VI1` is drastically efficient.

6 Integrating Point-based Improvement

In this section, we integrate a point-based improving procedure into value iteration over subspace and report our experiments on a larger POMDP problem.

6.1 Point-based improvement

The standard DP update $T\mathcal{V}$ is difficult because it has to account for infinite number of belief states. However, given a set $\mathcal{V}$ and a belief state b, computing the vector in the set $T\mathcal{V}$ at b is much easier. This can be accomplished by using a so-called *backup operator*.

Given a set $\mathcal{V}$ of vectors, a point-based procedure heuristically generates a finite set of belief points and backs up on the set to obtain a set of vectors. It is designed to have

this property: the value function represented by the set of backup vectors is better than the input set $\mathcal{V}$. Because the set of belief states are generated heuristically, point-based improvements is much cheaper than DP improvements.

A point-based value iteration algorithm interleaves standard DP update with multiple steps of point-based improvements. The standard DP update ensures that the output value function is ϵ-optimal when value iteration terminates.

6.2 Backup operator

In value iteration over subspace, DP update computes $\mathcal{V}_{n+1}^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ from $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$. To do so, it computes the set $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a',z')}$ for each $[a', z']$ pair. It is conceivable that this is still not so easy because $\phi(\mathcal{B}, a', z')$ usually consists of infinite number of belief states.

Consequently, it is necessary to design a point based procedure to improve $\mathcal{V}_n^{\phi(\mathcal{B},a',z')}$ before it is fed to DP update over subspace $\phi(\mathcal{B}, a', z')$. We can generate heuristically a finite set of belief states in the simplex and back up on this set to obtain a set of vectors. The key problem is, given a set $\mathcal{V}_n^{\phi(\mathcal{B},\mathcal{A},\mathcal{Z})}$ and a belief state b in $\phi(\mathcal{B}, a', z')$, how to compute a vector in the set $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a',z')}$? We can define a backup operator in this context. The backup vector can be built by three steps as follows.

1. For each action a and each observation z, find the vector in $\mathcal{V}_n^{\phi(\mathcal{B},a,z)}$ that has maximum inner product with $\tau(b, a, z)$. Denote it by $\beta_{a,z}$.
2. For each action a, construct a vector β_a by: for each s in the set $\mathcal{S}_{a'z'}$,

$$\beta_a(s) = r(s, a) + \gamma \sum_{z \in \mathcal{Z}} \sum_{s' \in \mathcal{S}_{az}} P(s', z|s, a)\beta_{a,z}(s')$$

   where $P(s', z|s, a)$ equals to $P(s'|s, a)P(z|s, a)$.
3. Find the vector, among the β_a's, that has maximum inner product with b. Denote it by β.

It can be proven that β is a vector in $\mathcal{V}_{n+1}^{\phi(\mathcal{B},a',z')}$. With the backup operator, before the set $\mathcal{V}_n^{\phi(\mathcal{B},a,z)}$ is fed to DP update over subspace, it is improved by multiple steps of point-based procedure. After these preliminary steps, the improved sets are fed to DP update over subspace. As such, we expect that the number of iterations can be reduced as value iteration converges.

6.3 Experiments

The problem is an extended version of the 3x3 grid world. It is illustrated in Figure 4. It has 35 columns. The goal location is 104, marked by $*$ in the figure. The agent is informed of its column number. So the problem has 105 states, 35 observations and 5 actions. The transition and observation models are similar to those in 3x3 grid world.

For simplicity, we use `PB-VI1` and `VI1` to refer to the algorithms conducting DP over subspace with and without integration of point-based procedure. Due to space

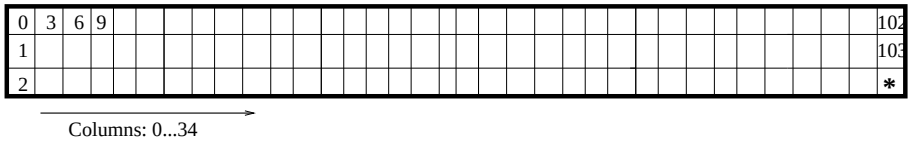


Fig. 4. A 3x35 grid world

limit, we report only on the time cost of DP update in `VI1` and `PB-VI1`. For convenience, the time reported for a DP update of `PB-VI1` consists of two portions: the time for multiple point-based improvements and the time for a DP update over subspace. The results are collected in Figure 5. Note that the time axis is drawn in log-scale.

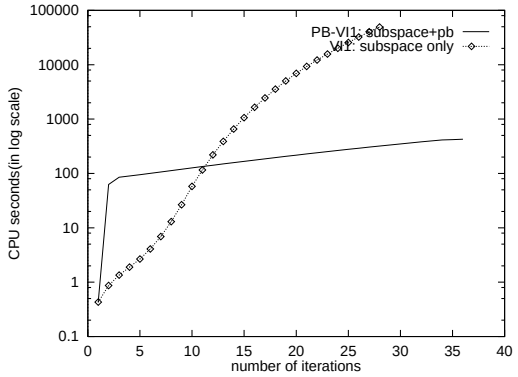


Fig. 5. Comparative study on value iteration over subspace with/o point-based improving

We see that `VI1` stand-alone is still insufficient to solve the problem. It takes little time in first 15 iterations but the DP time grows later on. For instance, for the 20th iteration, it takes 2,500 CPU seconds and for 27th iteration, it takes 9,000 seconds. It is believed that `VI` can by no means solve this problem.

The situation changes when the point-based procedure is integrated. `PB-VI1` converges in 425 seconds after 36 DP updates (plus point-based improvements) over belief subspace. On average, each iteration takes less than 15 seconds. It is much faster than `VI1`. In addition, `VI1` needs to run 207 iterations to converge for 3x3 grid world (mentioned in previous section). Taking this as a reference, we also note that the point-based procedure is still efficient in cutting down the number of iterations for `VI1` to converge.

7 Related Work and Future Directions

Our concept of informative POMDPs is very similar to that of regional observable POMDPs in [12]. Both of them assume that any observation restricts the world into a

small set of states. In [12], a regional observable POMDP is proposed to approximate an original POMDP and value iterations for regional observable POMDPs are conducted over the entire belief space. Our work focuses on accelerating value iterations for such POMDP class by restricting them over a subset of belief space.

The approach we use to exclude belief states from being considered works much like that in reachability analysis (e.g., see [6, 3]). In fully observable MDP, this technique is used to restrict value iteration over a small subset of state space. Even although value iteration is restricted into a subspace for informative POMDPs, we show that value function of good quality over entire belief space can be obtained from value functions over its subset. In addition, as mentioned in Subsection 4.5, the value function generated by value iteration over subspace is guaranteed to be ϵ-optimality without much effort.

Acknowledgments

This work has been supported by Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. HKUST658 / 95E).

References

1. Astrom, K. J.(1965). Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, 10, 403-406.
2. Bertsekas, D. P. and Gallagher, R. G.(1995). *Data Networks*. Prentice Hall., Englewood Cliffs, N. J..
3. Boutilier, C., Brafman, R. I. and Geib, C. (1998). Structured reachability analysis for Markov decision processes. In *Proceedings of UAI-98*.
4. Cassandra, A. R.(1998). *Exact and approximate algorithms for partially observable Markov decision processes*. PhD Thesis, Department of Computer Science, Brown University.
5. Cassandra, A. R., Littman, M. L. and Zhang, N. L.(1997). Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. *Proceedings of Thirteenth Conference on Uncertainty in Artificial Intelligence*, 54-61.
6. Dean, T., Kaelbling, L. P., Kirman, J. and Nicholson, A. (1995). Planning under time constraints in stochastic domain. *Artificial Intelligence*, volume 76, number 1-2, Pages 35-74.
7. Choi, S.P.M., Yeung, D. Y. and Zhang, N.L. *An environment model for non-stationary reinforcement learning*. Advances in Neural Information Processing Systems 12(NIPS-99), 987-993.
8. Hansen, E. A. (1998). *Finite-memory controls of partially observable systems.* PhD thesis, Depart of Computer Science, University of Massachusetts at Amherst.
9. Hauskrecht, M.(2000). Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 13, 33-94.
10. Papadimitriou, C. H. and Tsitsiklis, J. N.(1987). The complexity of Markov decision processes. *Mathematics of Operations Research*, Vol. 12, No. 3, 441-450.
11. Sondik, E. J. (1971). The optimal control of partially observable decision processes. Ph D thesis, Stanford University, Stanford, California, USA.
12. Zhang, N. L. and Liu, W. (1997). A model approximation scheme for planning in stochastic domains. *Journal of Artificial Intelligence Research*, 7, 199-230.
13. Zhang, N. L. and Zhang, W. (2001). Speeding up the convergence of value iteration in partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, Vol. 14, 29-51.

Improved Integer Programming Models and Heuristic Search for AI Planning

Yannis Dimopoulos

Department of Computer Science
University of Cyprus
CY-1678, Nicosia, Cyprus
yannis@cs.ucy.ac.cy

Abstract. Motivated by the requirements of many real-life applications, recent research in AI planning has shown a growing interest in tackling problems that involve numeric constraints and complex optimization objectives. Applying Integer Programming (IP) to such domains seems to have a significant potential, since it can naturally accommodate their representational requirements. In this paper we explore the area of applying IP to AI planning in two different directions.
First, we improve the domain-independent IP formulation of Vossen et al., by an extended exploitation of mutual exclusion relations between the operators, and other information derivable by state of the art domain analysis tools. This information may reduce the number of variables of an IP model and tighten its constraints. Second, we link IP methods to recent work in heuristic search for planning, by introducing a variant of `FF`'s enforced hill-climbing algorithm that uses IP models as its underlying representation. In addition to extending the delete lists heuristic to parallel planning and the more expressive language of IP, we also introduce a new heuristic based on the linear relaxation.

1 Introduction

Many recent successful approaches to AI planning, including planning graphs [2], propositional satisfiability [11] and heuristic search [3], are essentially restricted to planning domains representable in propositional logic. It seems however that many practical applications are beyond this representational framework, as they require expressing features like resources, numeric constraints, costs associated with actions, and complex objectives. Integer Programming (IP), and its underlying language of linear inequalities, seem to meet many of these requirements, at least from the representation perspective.

The study of the relevance of IP techniques to AI planning has only recently started to receive some attention. The `LPSAT` engine [16] integrates propositional satisfiability with an incremental Simplex algorithm; `Lplan` [6] uses the linear relaxation as a heuristic in a partial-order causal-link planner; `ILP-PLAN` [12] uses IP models for solving problems with resources, actions costs and complex objective functions; Bockmayr and Dimopoulos [5] show how IP models can be used to

incorporate strong forms of domain knowledge and represent compactly numeric constraints; finally, Vossen et al. [15] introduce a strong, domain-independent, method for translating STRIPS planning into IP models.

The IP models intend to enhance previous approaches to AI planning, like `BLACKBOX` or `GRAPHPLAN` , that essentially view planning as a constraint satisfaction problem. All these methods provide optimality guarantees for the solutions they generate, usually with respect to plan length. Recently, [13] and [3] introduced a new promising approach to AI planning that is based on heuristic search. Planners of this family, eg. [4, 10, 14], automatically extract heuristic functions from a planning problem specification and use it to guide the search for a solution in the state space. These planners do not provide any optimality guarantees, unless they employ admissible heuristics combined with optimal search algorithms, as eg. in [9].

In this paper we extend previous work on using IP in AI planning in two different directions. First, we improve the IP formulation of [15]. Second, we link IP methods to recent work in heuristic search for planning.

In the first part of the paper we describe an improved formulation of STRIPS planning, that exploits more fully the mutual exclusion relations between both the operators and the fluents, than this is done in [15]. Mutex information can strongly influence the way a planning problem is translated into inequalities, as it can yield formulations with fewer variables and constraints. Moreover, richer forms of information that can be derived automatically by domain analysis tools [7, 8], can further tighten the IP formulation of a planning domain.

In the second part of the paper we present a variant of `FF`'s [10] enforced hill-climbing algorithm that uses the IP models as its underlying representational language and is capable of generating parallel plans. Moreover, we introduce a new method for heuristic evaluation, that is based on solving the linear relaxation of the IP models, and compare it with the IP formulation of the delete lists relaxation method of `FF`.
As one may expect, `FF` clearly outperforms the new heuristic methods in term of running speed. This can be attributed partly to the fact that `FF` finds totally-ordered plans, while the new algorithms generate parallel plans. Moreover, while `FF` and similar planners, utilize highly optimized special-purpose techniques to speed-up heuristic evaluation, we rely on general purpose algorithms like Simplex and branch and bound. Nevertheless, generality has the advantage of extended expressiveness, as our approach can handle any domain representable in the language of linear inequalities. Moreover, we reiterate that the new algorithms generate parallel plans, a feature that can increase their usability in domains that are inherently parallel. This should be contrasted with most heuristic search based planners that generate totally-ordered plans, with the exception of recent work by Haslum and Geffner [9]. Additionally, the IP models can easily accommodate declarative domain knowledge and exploit it in the heuristic evaluation. Finally, IP formulations, combined with the linear relaxation heuristic allow us to implement a variety of strategies that offer a tradeoff between search time and solution quality.

The paper is organized as follows. Section 2 presents very briefly the IP formulation of [15] and then introduces the improved models. In section 3 we discuss new heuristic search methods based on the IP models. In section 4 we present and discuss some experimental results with the new IP formulation and the heuristic search algorithms, and in section 5 we conclude.

2 IP Models for Planning Problems

Integer Programming is a more general representation language than propositional logic. In order to represent a general linear inequality exponentially many clauses (in the number of variables) may be needed. Integer programming combines propositional logic with arithmetic and therefore allows for more compact formulations. On the other hand, any propositional clause $x_1 \vee \ldots \vee x_k \vee \bar{x}_{k+1} \vee \ldots \vee \bar{x}_{k+l}$ can be easily represented as a linear inequality $x_1 + \ldots + x_k - x_{k+1} - \ldots - x_{k+l} \geq 1 - l$ in 0-1 variables. However, such a straightforward translation usually leads to poor performance. Problem solving with IP techniques often requires a different translation of a problem into linear inequalities. Deriving a strong model for the problem to be solved is a fundamental issue for successfully applying IP.

As it is shown in [15] similar observations hold for AI planning. Instead of simply translating `SAT` encoding into linear inequalities, [15] presented a different, substantially stronger, domain-independent IP formulation of planning problems. A brief presentation of this approach follows (see [15] for details).

2.1 Domain Independent Modeling

Any STRIPS planning problem can be represented by a set of variables divided into action and state change variables. For each action a in the domain, we introduce an action variable $y_{a,i}$ which assumes the value true if a is executed at period i, and false otherwise. For each fluent f we define four variables, namely $x^{add}_{f,i}$, $x^{pre-add}_{f,i}$, $x^{pre-del}_{f,i}$, $x^{maintain}_{f,i}$. Variable $x^{maintain}_{f,i}$ encodes 'no-op' actions, while the other variables are defined as follows (symbol / denotes set difference).

$$\sum_{a \in pre_f / del_f} y_{a,i} \geq x^{pre-add}_{f,i}, \qquad y_{a,i} \leq x^{pre-add}_{f,i} \quad \forall a \in pre_f / del_f$$

$$\sum_{a \in add_f / pre_f} y_{a,i} \geq x^{add}_{f,i}, \qquad y_{a,i} \leq x^{add}_{f,i} \quad \forall a \in add_f / pre_f$$

$$\sum_{a \in pre_f \cap del_f} y_{a,i} = x^{pre-del}_{f,i}$$

Informally, $x^{add}_{f,i} = 1$ iff an action is executed at time point i that has f as an add effect but not as a precondition. Similarly, $x^{pre-add}_{f,i} = 1$ iff an action is executed at time point i that has f as a precondition but does not delete it, and $x^{pre-del}_{f,i} = 1$ if this action has f both as a precondition and a delete effect.
The constraints below prohibit parallel execution of mutually exclusive actions.

$$x_{f,i}^{add} + x_{f,i}^{maintain} + x_{f,i}^{pre-del} \leq 1$$

$$x_{f,i}^{pre-add} + x_{f,i}^{maintain} + x_{f,i}^{pre-del} \leq 1$$

The explanatory frame axioms are encoded as

$$x_{f,i}^{pre-add} + x_{f,i}^{maintain} + x_{f,i}^{pre-del} \leq x_{f,i-1}^{add} + x_{f,i-1}^{maintain} + x_{f,i-1}^{pre-add}$$

while the initial state constraints are represented by setting $x_{f,0}^{add}$ to 1 if f is true in the initial state and 0 otherwise. Finally, if $i \in 1,..,t$, for each goal f we add the constraint $x_{f,t}^{add} + x_{f,t}^{maintain} + x_{f,t}^{pre-add} \geq 1$.

2.2 Exploiting Domain Structure

The domain independent models of planning problems described above can be substantially improved by exploiting properties of the specific planning domain at hand. In particular, by analyzing in greater detail than in [15] the mutual exclusion relations between the operators and the fluents of a domain, we may be able to reduce the number of state-change variables and tighten the constraints of the IP model. We assume the availability of suitable domain analysis tools capable of identifying these relations, as those described eg. in [7,8]. We describe first, two improvements that aim at reducing the number of variables of the IP model.

- For each fluent f such that $\sum_{a \in add_f/pre_f} y_{a,i} \leq 1$, define $x_{f,i}^{add}$ as $x_{f,i}^{add} = \sum_{a \in add_f/pre_f} y_{a,i}$. This is a simple modification that allows us to substitute out variable $x_{f,i}^{add}$. Similar observations hold for the $x_{f,i}^{pre-add}$ variables.
- Let f be a fluent such that $y_{a_1,i} + y_{a_2,i} \leq 1$ holds for every pair of actions $a_1 \in pre_f/del_f$ and $a_2 \in add_f/pre_f$. Then merge $x_{f,i}^{pre-add}$ and $x_{f,i}^{maintain}$ substituting out variable $x_{f,i}^{pre-add}$. With this modification we can omit all constraints that refer to $x_{f,i}^{pre-add}$ but we need to include constraints of the form $y_{a,i} \leq x_{f,i}^{maintain}$ for all $a \in pre_f/del_f$ that reflect the new, extended, meaning of the $x_{f,i}^{maintain}$ variables.

The first improvement is straightforward, while the second deserves some further discussion. Since every $a_1 \in pre_f/del_f$ is mutually exclusive with every $a_2 \in add_f/pre_f$, the variables $x_{f,i}^{pre-add}$ and $x_{f,i}^{add}$, are also exclusive. Instead of adding an additional constraint, we omit variable $x_{f,i}^{pre-add}$ and "transfer" its role in the model to the corresponding variable $x_{f,i}^{maintain}$. Note that while exclusion constraints of the form $x_{f,i}^{pre-add} + x_{f,i}^{maintain} + x_{f,i}^{pre-del} \leq 1$ are dropped, the constraints $x_{f,i}^{add} + x_{f,i}^{maintain} + x_{f,i}^{pre-del} \leq 1$ remain in the formulation, marking $x_{f,i}^{add}$ and $x_{f,i}^{maintain}$ (and therefore the deleted $x_{f,i}^{pre-add}$) as mutually exclusive.

We now discuss some techniques that can further tighten the IP model of a planning domain. Our method is based on the derivation of *single-valuedness* and *XOR* constraints as described in [8]. We restrict our discussion to binary fluents. A single valuedness constraint is a constraint of the form $(f(y, *z), C(y))$ stating that for every value of variable y that satisfies constraint $C(y)$ there can be only one value for (the "starred") variable $*z$. An *XOR* constraint is a constraint of the form (XOR $f_1(y, z), f_2(y, u), C(y)$) stating that for every state and for every value of y satisfying $C(y)$ either $f_1(y, z)$ or $f_2(y, u)$ must be true (for some values of z and u) but not both.

- Let $f(y, z)$ be a binary fluent for which the single-valuedness constraint $(f(y, *z), C(y))$ holds. Then, for each y that satisfies $C(y)$, replace the set of constraints $x^{add}_{f,i} + x^{maintain}_{f,i} + x^{pre-del}_{f,i} \leq 1$ that refer to each possible value of z, with a single constraint

$$\sum_{z} x^{add}_{f,i} + \sum_{z} x^{maintain}_{f,i} + \sum_{z} x^{pre-del}_{f,i} \leq 1$$

  where $\sum_{z}$ denotes the sum over the domain of the second parameter of the fluent f, namely z.

Going one step further we can exploit the *XOR* constraints and modify the mutual exclusion constraints as follows.

- Let $f_1(y, *z)$ and $f_2(y, *u)$ be two single-valued fluents on their second arguments, for which the constraint (XOR $f_1(y, z), f_2(y, u), C(y)$) holds. Then, replace all mutual exclusion constraints on f_1 and f_2 that refer to some specific object satisfying $C(y)$ with the constraint

$$\sum_{z} x^{add}_{f_1,i} + \sum_{z} x^{maintain}_{f_1,i} + \sum_{z} x^{pre-del}_{f_1,i} + \sum_{u} x^{add}_{f_2,i} + \sum_{u} x^{maintain}_{f_2,i} + \sum_{u} x^{pre-del}_{f_2,i} \leq 1$$

  Moreover, if the model does not contain any of the variables $x^{pre-add}_{f_1,i}$ and $x^{pre-add}_{f_2,i}$ (meaning that all actions that have f_1 or f_2 as a precondition, also have it as a delete effect), we can replace $\leq$ in the above constraint with an equality.

The following simplification relates to the frame axioms.

- Let f be a fluent such that every action that adds it, has f as its sole add effect. Then, if f is true at time $i-1$ we can safely add the constraint

$$x^{add}_{f,i-1} + x^{maintain}_{f,i-1} + x^{pre-add}_{f,i-1} \leq x^{pre-add}_{f,i} + x^{maintain}_{f,i} + x^{pre-del}_{f,i}$$

  which combined with the corresponding explanatory frame axiom, namely

$$x_{f,i}^{pre-add} + x_{f,i}^{maintain} + x_{f,i}^{pre-del} \leq x_{f,i-1}^{add} + x_{f,i-1}^{maintain} + x_{f,i-1}^{pre-add}$$

gives rise to an equality of the form

$$x_{f,i}^{pre-add} + x_{f,i}^{maintain} + x_{f,i}^{pre-del} = x_{f,i-1}^{add} + x_{f,i-1}^{maintain} + x_{f,i-1}^{pre-add}$$

To see that the above transformation is valid, note that if a fluent f is true at time $i-1$ (meaning that one of the $x_{f,i-1}^{add}$, $x_{f,i-1}^{maintain}$, $x_{f,i-1}^{pre-add}$ is true), and all actions that add f have no other add effects, then assigning false to $x_{f,i}^{add}$, does not have unwanted complications. In fact, if we adopt the above simplification, $x_{f,i}^{add}$ will necessarily be assigned false, if f is true at time $i-1$ and f does not have an associated variable $x_{f,i}^{pre-add}$ or this variable is assigned the value false. We note that the above transformation of the frame axioms into equalities can be extended to more general cases, but we do not discuss this issue further.

Example: Consider the rocket domain with the usual *load* and *unload* operators for packages and fly for airplanes. We note that all actions that add *in* for packages are mutually exclusive, therefore the fluent variables $x_{in,i}^{add}$ can be substituted out and replaced by $\sum_{a \in add_{in}/pre_{in}} y_{a,i}$, where a is a *load* action that adds the corresponding *in* proposition. Similar constraints hold for *at* for both planes and packages, hence all corresponding $x_{f,i}^{add}$ can be omitted from the formulation. Now consider the variable $x_{at,i}^{pre-add}$ corresponding to the fluent *at* that refers to airplanes. Note that every action $a_i \in pre_{at}/del_{at}$ (ie. load and unload actions) is mutually exclusive with every action $a_j \in add_{at}/pre_{at}$ (ie. fly actions) and therefore we can merge $x_{at,i}^{pre-add}$ with $x_{at,i}^{maintain}$, by omitting all $x_{at,i}^{pre-add}$ and adding the constraints $y_{ld,i} \leq x_{at,i}^{maintain}$ and $y_{un,i} \leq x_{at,i}^{maintain}$ for the corresponding load (denoted as $y_{ld,i}$) and unload ($y_{un,i}$) actions.
Moreover the single-valuedness of $in(x, *y)$, where x refers to packages and y to planes, will tighten the mutual exclusion constraint $\sum_L y_{un,i} + x_{in,i}^{maintain} + \sum_L y_{ld,i} \leq 1$ into the stronger constraint

$$\sum_{Pl} \sum_{L} y_{un,i} + \sum_{Pl} x_{in,i}^{maintain} + \sum_{Pl} \sum_{L} y_{ld,i} \leq 1$$

where $\sum_{Pl}$ denotes sum over all planes and $\sum_L$, sum over all locations. A similar constraint can be derived for fluent *at* that refers to packages being at locations. Since fluents *in* and *at* are related with a XOR constraint (meaning that, at each time, a package must be *in* some plane, or *at* some location but not both) we can combine the two constraints and derive

$$\sum_{Pl} \sum_{L} y_{un,i} + \sum_{Pl} x_{in,i}^{maintain} + \sum_{Pl} x_{at,i}^{maintain} + \sum_{Pl} \sum_{L} y_{ld,i} = 1$$

Finally, since the only operator that adds *at* for a package and a location is *unload*, and *at* is the only add effect of this operator, the corresponding frame

axioms can be converted into equalities. Similar observations hold for the other propositions of the domain.

In the next section, when we introduce the constraint relaxation heuristic, we will need IP models for domains with operators that do not contain delete effects. For this special case we use a straightforward translation of propositional satisfiability planning theories into linear inequalities.

3 Heuristic Search

The above modifications in the IP formulation of planning problems, can substantially improve performance. However, as it happens with other approaches that generate optimal plans, in many domains, IP models do not scale well. In this section, we attempt to address this issue in a flexible way, by bringing together IP modeling and heuristic search. Heuristic search methods derive a heuristic function from the problem specification and use it to guide the search in the state space. The heuristic function h for a problem P is derived by considering a relaxed problem P'.

We consider two different relaxations of a planning problem. The first, which we call the *constraint relaxation* (CL) approach, was introduced in [3] and modified in `FF` [10]. Here the relaxed problem is obtained from the original by ignoring the delete effects of the operators. In the second approach, which we call *linear relaxation* (LR) approach, the relaxed problem is obtained from the original problem by dropping the integrality constraint from the integer variables.

The heuristic function we use is the same as in `FF`. Let $O_1, O_2, ..., O_m$ be the sets of actions selected (action selection can be a fractional number greater than 0, if the LR approach is used) in the solution of the relaxed problem at time i. Variable m, called the *length invariant*, equals the number of time steps needed so that the relaxed problem becomes solvable (ie., the relaxation with $m-1$ time steps is infeasible). We define our heuristic function as $h(S) = \sum_{i=1,...,m} |O_i|$.

The search method we use in our approach is a variant of the enforced hill-climbing introduced in `FF`. It can be described briefly as follows.

Algorithm **MEHC**(*step*,*nd-limit*,*cutoff*)
i:=0; solved:=false;
while not solved
  i:=i+1; Solve the relaxed problem using i time steps;
  if feasible **then** solved:=true;
endwhile
m:=i; /*Length invariant m used in the heuristic function*/
Set *obj* to the value of the objective function and current plan to empty;
S :=Initial State; $h(S) = obj$;
while $h(S) \neq 0$ **do**
  Call **EBFS**(*m*,*step*,*nd-limit*,*cutoff*) in order to
  breadth-first search for a state S' with $h(S') < h(S)$;
  if (no such state is found) **then** report failure and stop;

Add the selected actions to current plan, set $S := S'$ and $h(S) := h(S')$;
endwhile

Algorithm `MEHC` differs from `FF` in the way it performs the search for the successor state at each of its iterations. The new search method is implemented by procedure `EBFS`, which is presented below.

For a planning problem P, let P_t denote the set of constraints in the IP formulation of P over the time interval t. Moreover, let P_t^{lr} denote the set of constraints obtained if the integrality constraints of P_t are dropped. Finally, let P_t^{cr} denote the set of constraints, over the time interval t, of the IP model of the problem obtained from P by dropping the delete list of the operators. Then, procedure `EBFS` below implements the breadth-first search for a state with a better heuristic value, where P_t' stands for any of P_t^{lr} and P_t^{cr} depending on the method we employ.

procedure **EBFS**(m,*step*,*nd-limit*,*cutoff*)
i=*step*;
while (i<*cutoff*+*step*)
set branch and bound node limit to *nd-limit*, and
solve $\min(\sum_{a \in A} \sum_{j \in [i+1,m]} y_{a,j})$ subject to
$P_{[0,i]} \cup P'_{[i+1,m]} \cup \{\sum_{a \in A} \sum_{j \in [i+1,m]} y_{a,j} < h(S)\}$;
if (feasible) **then** return solution **else** i:=i+1;
endwhile

Note that the set of constraints $P_{[0,i]}$ allows for parallel action execution, and therefore, the successor of a state S can be any state that can be reached from S by executing a *set* of parallel actions. Hence, the algorithm generates *parallel plans*.

Procedure `EBFS` uses branch and bound in order to perform the search for the successor state, and therefore its theoretical time complexity is determined mainly by the number of integer variables of the problem it solves. If the linear relaxation heuristic is used, each iteration of `EBFS` has time complexity which is, in the worst case, exponential in the number of variables of $P_{[0,i]}$. In the case of the constraint relaxation heuristic, this complexity is higher, as it is exponential in the number of variables of $P_{[0,i]} \cup P^{cr}_{[i+1,m]}$. However, our experimentation revealed that, in many domains, the set of constraints P_t^{lr} is much harder to satisfy than the corresponding set P_t^{cr}. Consequently, the constraint relaxation heuristic can outperform the linear relaxation one in terms of running speed.

Moreover, the use of branch and bound in `EBFS` has some interesting implications. For instance, in the case of the constraint relaxation approach, the algorithm can branch on any of the variables of $P_{[0,i]} \cup P^{cr}_{[i+1,m]}$, interleaving in this way the selection of the successor state with its evaluation. Furthermore, the objective function, which minimizes the number of actions, can provide substantial guidance in the search of a low cost successor state.

The new algorithm is parametric to the values of *step*, *nd-limit*, and *cutoff*. Parameter *nd-limit* defines the node limit of the branch and bound search algorithm. Parameter *step* defines the minimum distance (number of parallel steps) of the successor state from the current state, while *cutoff* defines the maximum such distance. Parameters *step* and *nd-limit* allow us to implement a variety of search strategies that trade-off solution quality for performance. Higher values for *nd-limit* may generate successor states with better heuristic values, while higher values for *step* usually lead to more informed choices in the selection of the successor state. Therefore, higher values for these parameters usually yield better plans, while lower values better run times.

4 Experimental Results

We run some initial experiments with the new IP formulation and the heuristic search method on a variety of planning domains. The models were generated by hand, using the algebraic modeling system `PLAM` (ProLog and Algebraic Modeling) [1] and following the steps described in section 2. In all the experiments `CPLEX` 6.6 was used. All variables were declared integer. The setting was the following. At each node of the branch and bound dual simplex with "steepest-edge pricing" was used. Probing was set to 1, leading to some simplifications of the models, as well as clique cuts derivation. The variable selection strategy was set to "pseudo-reduced costs", and the node selection strategy to best-bound search. All experiments were run on a Sun Ultra-250 with 512 MB RAM.

Table 1 compares the performance of the IP formulation of [15] (column OIP) and the improved formulation discussed in section 2 (column IIP) on blocks world and rocket domain problems. The objective function in both domains was set to minimize the number of actions. In the rocket domain, some flight minimization experiments were also run, and are marked with the problem name suffix `fl-min` in Table 1. The entries under "First" refer to the run time and number of nodes explored until the first solution was found. For the blocks world problems the entries under "Optimal" refer to solving these problems to optimality (the time needed to prove optimality is included). The same entries for the, more difficult, rocket problems refer to finding a solution with cost that is provably not more than 10% higher than the cost of the optimal solution. The data of Table 1 were obtained using, in each domain and for each formulation, the cut generation strategy that seems to perform best. In the blocks world domain the default values were used for both formulations. In the rocket domain, different Clique and Gomory cut generation strategies were used for the different formulations and the different optimization objectives, but we do not discuss this issue further.

Both domains allow for parallel actions. The largest problem in the blocks world domain, `bw3`, involves 13 blocks, has plan length 9 and the optimal solution has 19 actions. The IP formulation for this domain is strong, leading to small integrality gaps and, consequently, good performance. In some moderately sized problems, the first solution was obtained by simply rounding the values of the variables obtained after solving the linear relaxation.

	First				Optimal			
	OIP		IIP		OIP		IIP	
Problem	time	nodes	time	nodes	time	nodes	time	nodes
bw1	93	5	25	0	93	5	28	11
bw2	310	0	108	0	310	0	108	0
bw3	7072	21	876	23	-	-	1277	50
rocket1	98	371	13	56	631	2554	92	287
rocket2	109	297	10	28	1521	4237	132	260
rocket3	-	-	408	459	-	-	1885	1251
rocket4	7478	2522	461	456	-	-	1886	993
rocket1-fl-min	1828	1734	19	14	3775	5417	59	422
rocket2-fl-min	591	384	80	320	3602	4754	82	360
rocket3-fl-min	-	-	236	260	-	-	1685	4791
rocket4-fl-min	3391	372	210	150	-	-	2083	4067

Table 1. Performance comparison of different IP formulations on action and flight (marked with the name suffix `fl-min`) minimization problems. For each problem we give run time and number of nodes in the branch and bound tree. Times in seconds. A dash denotes that no solution was found within about 2 hours (8000 sec) of CPU time.

In the rocket domain, the largest problem, `rocket4`, involves 16 packages, 5 locations and 3 planes. The optimal parallel plan length is 7, and the optimal solution contains 41 actions. Here the IP formulation is weaker. The integrality gap is larger, and closes relatively slow, after many iterations. Nevertheless, in most cases, the new formulation is substantially stronger.

Table 2 shows some representative results from experiments with the IP based heuristic methods on the parallel rocket domain. The CR columns refer to the constraint relaxation heuristic, while the LR columns to the linear relaxation one. The `FF` column shows the number of actions in the plan generated by `FF`, which solves all problems in a few seconds.

				LR			CR			FF
Problem	pac	pl	loc	time	len	actions	time	len	actions	actions
rocket4	16	3	5	29	9	45	25	13	46	53
rocket5	16	3	5	62	9	47	36	14	45	41
rocket6	16	3	5	33	9	43	31	13	49	52
rocket7-1	21	3	6	445	11	57	27	11	56	67
rocket7-2	21	3	6	427	12					
rocket7-3	21	3	6	286	12					
rocket8	27	4	7	1598	11	75	214	13	90	80

Table 2. Performance comparison of the heuristic search algorithms. For each problem we give the number of packages (pac), planes (pl) and locations (loc), solution time in seconds (time), parallel plan length (len) and number of actions in the plan (actions).

In all problems the relaxation based algorithm was run with the step and cutoff parameters set to 1. For the smaller problems, `rocket4` to `rocket6`, the *nd-limit* parameter of the linear relaxation based algorithm was set to infinity, ie. at each iteration the corresponding problem was solved to optimality. However, for larger problems, like `rocket7` and `rocket8`, when the linear relaxation heuristic is used, the number of explored nodes has to be limited in order to gain acceptable efficiency. The 3 entries of Table 2 prefixed with `rocket7`, correspond to solutions of the same problem with different node bounds. Row `rocket7-1` refers to solving the problem without limiting the number of explored nodes, while rows `rocket7-2` and `rocket7-3` correspond to a limit of 1000 and 500 nodes respectively. For problem `rocket8` the node limit was set to 500. In the problems we considered, restricting the number of nodes affects only the first two iterations of the algorithm, as in the subsequent iterations the optimal solution is found after exploring a few nodes. More specifically, in most problems, the first two iterations make up more than 60% of the total solution time.

In most of the problems it seems that the linear relaxation heuristic offers a reasonably good trade-off between search time and solution quality. For instance, if we compare the solution time of the direct IP formulation of problem `rocket4` in Table 1, with the time needed for solving the same problem with the linear relaxation heuristic algorithm in Table 2, we note a decrease from 461 secs to 29 secs. This speed-up comes with an increase of the plan length from 7 to 9.

It is interesting to compare the characteristics of the linear and constraint relaxation methods. The constraint relaxation heuristic almost always runs faster than the linear relaxation one. But as far as parallel plan length is concerned, constraint relaxation is quite unstable, and in many cases (eg. problems `rocket5` and `rocket8` in Table 2) generates plans that underutilize the available resources (planes) and, for this reason, are longer.

5 Conclusions and discussion

The results presented in [15], suggest that careful modeling can make IP effective in solving classical STRIPS problems. This has important practical implications, since many problems can be represented as a set of STRIPS operators together with some additional complex constraints. Solving such problems effectively, requires reasonable performance on their STRIPS part.

In this paper we presented some improvements of the IP formulation of [15] that exploit more fully the structure of the planning domains. The new translation method benefits from recent work in automated domain analysis, but also recent advances in Integer Programming. Indeed, advanced features of state-of-the-art IP solvers, such as preprocessing, probing, and constraint derivation, most notably in the form of Gomory and Clique cuts, have positive effect on the performance of the models. Our current work focuses on further improving the IP formulation of planning problems, and combining it with strong forms of domain knowledge. More extensive experimentation, to be reported in a longer version of this paper, gives encouraging first results.

The enforced hill-climbing algorithm that we described in the second part of the paper, can be understood as an attempt towards combining IP models with heuristic search. This is done in a way different than in the `Lplan` system [6], which uses the linear relaxation of an IP formulation, which is different than ours, as a heuristic in a partial-order causal-link planner.

Our intention is to develop algorithms that improve efficiency at an acceptable cost in solution quality. In parallel domains, which is our main focus, the degree of parallelism of the generated plan is an integral part of solution quality. It seems that the linear relaxation heuristic performs better than the constraint relaxation in terms of solution quality, but it is slower. We currently work on improving its performance.

Obviously, the algorithm we presented is neither complete nor optimal. Future research will focus on investigating whether it is feasible to employ IP models in heuristic search algorithms that satisfy both properties.

References

1. P. Barth and A. Bockmayr. Modelling discrete optimisation problems in constraint logic programming. *Annals of Operations Research*, 81:467–496, 1998.
2. A. Blum and M. Furst. Fast Planning Through Planning Graph Analysis. *IJCAI-95*, pp. 1636-1642, 1995.
3. B. Bonet, G. Loerincs and H. Geffner. A fast and robust action selection mechanism for planning. *AAAI-97*, pp. 714-719, 1997.
4. B. Bonet and H. Geffner. Planning as heuristic search: New Results. *ECP-99*, pp. 360-372, 1999.
5. A. Bockmayr and Y. Dimopoulos. Integer Programs and Valid Inequalities for Planning Problems. *ECP-99*, pp. 239-251, 1999.
6. T. Bylander. A Linear Programming Heuristic for Optimal Planning. *AAAI-97*, pp. 694-699, 1997.
7. M. Fox and D. Long. The Automatic Inference of State Invariants in TIM. *Journal of AI Research*, 9, pp. 367-421, 1998.
8. A. Gerevini and L. Schubert. Discovering state constraints in `DISCOPLAN`: Some new results. *AAAI-00*, pp. 761-767, 2000.
9. P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. *AIPS-00*, pp. 140-149, 2000.
10. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of AI Research*, Vol. 4, pp. 253-302, 2001.
11. H. Kautz and B. Selman. Unifying SAT-based and Graph-based Planning. *IJCAI-99*, pp. 318-325, 1999.
12. H. Kautz and J. Walser. State-space Planning by Integer Optimization. *AAAI-99*, pp. 526-533, 1999.
13. D. McDermott. A Heuristic Estimator for Means-Ends Analysis in Planning. *AIPS-96*, pp. 142-149, 1996.
14. I. Refanidis and I. Vlahavas. GRT: A Domain Independent Heuristic for STRIPS Worlds based on Greedy Regression Tables. *ECP-99*, pp. 346-358, 1999.
15. T. Vossen, M. Ball, A. Lotem, and D. Nau. On the Use of Integer Programming Models in AI Planning. *IJCAI-99*, pp. 304-309, 1999.
16. S. Wolfman and D S. Weld. The LPSAT Engine and Its Application to Resource Planning. *IJCAI-99*, pp. 310-317, 1999.

RIFO Revisited: Detecting Relaxed Irrelevance

Jörg Hoffmann and Bernhard Nebel

Institute for Computer Science
Albert Ludwigs University
Georges-Köhler-Allee, Geb. 52
79110 Freiburg, Germany
hoffmann@informatik.uni-freiburg.de

Abstract. RIFO, as has been proposed by Nebel et al. [8], is a method that can automatically detect irrelevant information in planning tasks. The idea is to remove such irrelevant information as a pre-process to planning. While RIFO has been shown to be useful in a number of domains, its main disadvantage is that it is not completeness preserving. Furthermore, the pre-process often takes more running time than nowadays state-of-the-art planners, like FF, need for solving the entire planning task.
We introduce the notion of relaxed irrelevance, concerning actions which are never needed within the relaxation that heuristic planners like FF and HSP use for computing their heuristic values. The idea is to speed up the heuristic functions by reducing the action sets considered within the relaxation. Starting from a sufficient condition for relaxed irrelevance, we introduce two preprocessing methods for filtering action sets. The first preprocessing method is proven to be completeness-preserving, and is empirically shown to terminate fast on most of our testing examples. The second method is fast on all our testing examples, and is empirically safe. Both methods have drastic pruning impacts in some domains, speeding up FF's heuristic function, and in effect the planning process.

1 Introduction

RIFO, as has been proposed by Nebel et al. [8], is a method that can automatically detect irrelevant information in planning tasks. A piece of information can be considered irrelevant if it is not necessary for generating a solution plan. The idea is to remove such irrelevant information as a pre-process in the hope to speed up the planning process. While RIFO has been shown to be useful for speeding up GRAPHPLAN in a number of domains, it does not guarantee that the removed information is really irrelevant. In effect, RIFO is not completeness preserving. Furthermore, the pre-process itself can take a lot of running time. While RIFO can be proven to terminate in polynomial time, it—or at least its implementation within IPP4.0 [7]—is on a lot of planning tasks not competitive with nowadays state-of-the-art planners. In our experiments on a large range of tasks from different domains, we found that in most examples RIFO needs more running time to finish the pre-process than FF needs for solving the entire task.

In this paper, we present a new approach towards defining and detecting irrelevance. We explore the idea of *relaxed irrelevance*, which concerns pieces of information, precisely STRIPS actions, that are not needed within the relaxation that state-of-the-art heuristic planners like FF [4] and HSP [2] use for computing their heuristic values. Those planners evaluate each search state S by estimating the solution length from S under the relaxation that all delete lists are ignored. The main bottleneck in FF and HSP is the heuristic evaluation of states, so it is worthwhile trying to improve on the speed of such evaluations. Our idea is to speed

up the heuristic functions by reducing the action sets considered within the relaxation. Actions that are relaxed irrelevant need never be considered. We define the notion of *legal generation paths*, and prove that an action is relaxed irrelevant if it does not start such a path. Deciding about legal generation paths is still NP-hard, so we introduce two approximation techniques. Both can be used as preprocessing methods for filtering the action set to be considered within the relaxation. The first preprocessing method includes all actions that start a legal generation path, and can therefore safely be applied to the relaxation. The pre-process terminates fast on most of our testing examples in the sense that it is orders of magnitude faster than FF. The second approximation method is fast on all our testing examples, and while it is not provably completeness preserving, it is empirically safe: from a large testing suite, no single example task got unsolvable because of the filtering process.

We introduce our theoretical investigations and algorithmic techniques within the STRIPS framework, and summarise how they are extended to deal with conditional effects. Both action filtering methods can in principle be used as a pre-process to either FF or HSP—or rather as a pre-process to any planner that uses the same relaxation—and both methods have drastic pruning impacts in some domains. We have implemented the methods as a pre-process to FF, and show that they significantly speed up FF's heuristic function, and in effect the plan generation process, in those cases where the pruning impact is high.

The next section gives the necessary background in terms of STRIPS notations and heuristic forward state space planning as done by FF and HSP. Section 3 defines and investigates our notions of relaxed irrelevance and legal generation paths.[1] Section 4 explains two ways of approximating legal generation paths, yielding the above described two action filtering methods. Section 5 summarises how our analysis is extended to ADL domains, and Section 6 describes the experiments we made for evaluating the approach. Section 7 explains two lines of work that we are currently exploring. Section 8 concludes.

2 Background

We introduce our theoretical observations and our algorithms in a propositional STRIPS framework, where planning tasks are triples $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ comprising the action set, the initial state, and the goal state, actions are triples $o = (\mathrm{pre}(o), \mathrm{add}(o), \mathrm{del}(o))$, and the result of applying an action o to a state S with $\mathrm{pre}(o) \subseteq S$ is $\mathit{Result}(S, o) = (S \cup \mathrm{add}(o)) \setminus \mathrm{del}(o)$. Plans, or solutions, are sequences P of actions for which $\mathcal{G} \subseteq \mathit{Result}(\mathcal{I}, P)$ holds. A plan $P = \langle o_1, \ldots, o_n \rangle$ is called *minimal*, if no single action can be left out of the sequence without loosing the solution property, i.e., if $\langle o_1, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n \rangle$ is not a solution for any o_i. The *length* of a plan is the number of actions in the sequence. A plan for a task $\mathcal{P}$ is *optimal* if it has minimal length among all plans for $\mathcal{P}$. Obviously, optimal plans are minimal.

FF is based on the general principle of heuristic forward state space search, as has first been implemented in HSP1.0. The idea is to search in the space of states that are reachable from the initial state, trying to minimise a heuristic value that is computed to each considered state. The heuristic evaluation in both FF and HSP is based on the following relaxation.

Definition 1. *Given a planning task* $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$. *The* relaxation $\mathcal{P}'$ *of* $\mathcal{P}$ *is defined as* $\mathcal{P}' = (\mathcal{O}', \mathcal{I}, \mathcal{G})$, *with*

$$\mathcal{O}' = \{(\mathit{pre}(o), \mathit{add}(o), \emptyset) \mid (\mathit{pre}(o), \mathit{add}(o), \mathit{del}(o)) \in \mathcal{O}\}$$

[1] We only sketch our proofs. The complete proofs can be found in a longer version of the paper, available as a technical report [5].

In words, a planning task is relaxed by ignoring all delete lists. When either FF or HSP face a search state S, they estimate the length of a relaxed solution starting in S, i.e., they estimate the solution length of the task $(\mathcal{O}', S, \mathcal{G})$. In HSP, this is done by computing certain weight values for all facts, where the weight of a fact is an estimate of how difficult it is to achieve that fact from S. Computing these weight values involves a fixpoint computation that iteratively applies all actions until no more changes occur [2]. In FF, the solution length to $(\mathcal{O}', S, \mathcal{G})$ is estimated by extracting an explicit solution in a GRAPHPLAN-style manner [1, 4]. The technique is based on building a relaxed version of GRAPHPLAN's planning graph, which involves, like HSP's method, repeated application of all actions.

The main bottleneck in HSP, i.e., the main source of running time consumed, is the heuristic evaluation of states [2]. The same applies to FF. While heuristic evaluation is implemented efficiently in both systems, usually no more than a few hundred state evaluations can take place in a second (for FF, Section 6 provides averaged running times per state evaluation on a large range of domains). In some huge planning tasks, we have observed that a single evaluation in FF can take up to half a second running time. This is due to the large number of actions that there are in instantiated planning tasks. With ten-thousands of actions to be considered, FF's process of building a relaxed planning graph, and HSP's process of computing a weight fixpoint, must be costly no matter how efficient the implementation is. Our idea, consequently, is to reduce the number of actions that the planners need to consider within the relaxation, i.e., to compute as a pre-process a set $\mathcal{O}|_r$ of actions that are considered relevant for the relaxation. During search, one can then estimate solution lengths to the tasks $(\mathcal{O}|'_r, S, \mathcal{G})$ as opposed to using the whole action set in the tasks $(\mathcal{O}', S, \mathcal{G})$.

Of course, the set $\mathcal{O}|_r$ can not be chosen arbitrarily small. If important actions are missed out, then the task $(\mathcal{O}|'_r, S, \mathcal{G})$ can become unsolvable for a state S though it would be solvable with the original action set. In other words, one runs the risk of loosing relaxed completeness. If the task $(\mathcal{O}|'_r, S, \mathcal{G})$ is unsolvable, which both HSP's and FF's algorithmic methods will detect, then the systems set the heuristic value of S to ∞, excluding the state from the search space. While this is normally justified—if a state can not be solved even when ignoring delete lists, then that state is unsolvable—it can lead to incompleteness if solving $(\mathcal{O}|'_r, S, \mathcal{G})$ only failed because $\mathcal{O}|_r$ does not contain some important action(s).[2] The rest of the paper is inspired by a notion of relevance that maintains relaxed completeness.

3 Relaxed Irrelevance

We consider an action relaxed irrelevant if it never appears in an optimal relaxed solution. Clearly, such actions can be ignored within the relaxation without loosing completeness. Unfortunately, deciding about relaxed irrelevance is as hard as planning itself.

Definition 2. *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task. An action $o \in \mathcal{O}$ is* relaxed irrelevant *if o is not part of any optimal relaxed solution from any reachable state.*

Definition 3. *Let RELAXED-IRRELEVANCE denote the following problem:*
Given a planning task $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ and an action $o \in \mathcal{O}$, is o relaxed irrelevant?

[2] One might argue that this could be fixed by setting the heuristic value of S to a large integer instead of ∞. While this would regain completeness, it would also make the adequacy of the heuristic questionable: If a large number of states have the same high heuristic evaluation only because $\mathcal{O}|_r$ is too restrictive, then the heuristic is not very informative about the real structure of the search space.

Theorem 1. *Deciding RELAXED-IRRELEVANCE is PSPACE-hard.*

Proof Sketch: By a polynomial reduction from PLANSAT, the decision problem of whether there exists a solution plan for a given arbitrary STRIPS planning task [3]: First rename all atoms in the original task. Then put original o into the renamed action set, plus two artificial actions: one requiring the renamed goal to be solved, deleting all renamed atoms, and adding o's precondition, the other needing o's adds, and achieving the renamed goal. o is needed for an optimal relaxed solution in the modified task if and only if the original task is solvable. ∎

3.1 A Sufficient Condition

We now derive a sufficient condition for relaxed irrelevance. The following definition forms the heart of our investigation.

Definition 4. *Let $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task. The* generation graph *to the task is defined by the node set $\mathcal{O} \cup \{o_G\}$, with $o_G := (\mathcal{G}, \emptyset, \emptyset)$, and the edge set*

$$\{(o, o') \mid add(o) \cap pre(o') \neq \emptyset\}$$

We refer to paths $P = \langle o_1, \ldots, o_n = o_G \rangle$ in this graph as generation paths. *We call $add(o_i) \cap pre(o_{i+1})$ the* connecting facts *at position i. P is* legal *if at each position there is at least one connecting fact that is not contained in the preconditions of the previous actions, i.e., if for $1 \leq i \leq n-1$:*

$$(add(o_i) \cap pre(o_{i+1})) \setminus \bigcup_{1 \leq j \leq i} pre(o_j) \neq \emptyset$$

The generation graph to a task intuitively represents all ways in which facts can be achieved. A generation path is a sequence of actions that support each other, and that end up making at least one goal true. We will see in the following that the only generation paths that are adequate in minimal relaxed solutions are those generation paths that are legal. Precisely, we will show the following.

Theorem 2. *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task, S a state, and $P = \langle o_1, \ldots, o_n \rangle$ a minimal relaxed solution to S. Then for all o_i there exists a legal generation path P_i starting with o_i.*

With that, we immediately have our sufficient condition.

Corollary 1. *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task, $o \in \mathcal{O}$. If there is no legal generation path P starting with o, then o is not part of any minimal relaxed solution from any state. In particular, o is then relaxed irrelevant.*

Semantically, Definition 4 can be seen as a modification of the base technique that is used in RIFO. The relation between the techniques gives a nice picture of what is happening. Briefly, it can be explained as follows. To create an expectation of what is relevant for solving a planning task, RIFO builds a so-called fact-generation tree. This is an AND- OR-tree that is built by backchaining from the goals. The root node is an AND-node corresponding to the goals. Other AND-nodes correspond to an action's preconditions, and the OR-nodes are single atoms that can alternatively be achieved by different actions. Once this tree is generated, RIFO applies a number of simple heuristics to select the information from the tree that is likely most relevant. Now, the set of all legal generation paths can be viewed as a more restrictive version of RIFO's fact-generation tree, where an action is only allowed to achieve an OR-node if the intersection of the action's precondition with the facts on the path from the OR-node to the tree root is empty. This is adequate

(only) for relaxed planning. While RIFO selects fractions of its tree as relevant, we select the whole tree. This gives us completeness in the relaxation. The proof to Theorem 2 proceeds using what we call the *needed facts*, which are the facts for whose achievement actions can be placed at a certain position in a relaxed solution.

Definition 5. *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task, S a state, and $P = \langle o_1, \ldots, o_n \rangle$ a relaxed solution to S. The* open facts $OF(P, i)$ of P at position i *are*

$$OF(P,i) := (\mathcal{G} \setminus \bigcup_{i<j\leq n} add(o_j)) \cup \bigcup_{i<j\leq n} (pre(o_j) \setminus \bigcup_{i<k<j} add(o_k)),$$

and the needed facts $NF(P, i)$ of P at position i *are*

$$NF(P,i) := OF(P,i) \setminus (S \cup \bigcup_{1\leq j<i} add(o_j))$$

An action placed at position i in a relaxed plan P must add all needed facts of P at position i, and in a minimal relaxed plan there is at least one needed fact at each position.

Lemma 1. *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task, S a state, and $P = \langle o_1, \ldots, o_n \rangle$ a relaxed solution to S. Then $add(o_i) \supseteq NF(P, i)$ holds for $1 \leq i \leq n$.*

Proof Sketch: If an action does not add a needed fact, then P is no relaxed solution, because either some precondition ahead or some goal remains unachieved. ■

Lemma 2. *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task, S a state, and $P = \langle o_1, \ldots, o_n \rangle$ a minimal relaxed solution to S. Then $NF(P, i) \neq \emptyset$ holds for $1 \leq i \leq n$.*

Proof Sketch: If there is no needed fact at position i, then P without o_i is still a relaxed solution—all facts that must be achieved are true without applying o_i. ■

Using the above two lemmata, Theorem 2 can be proven, stating that to all actions o_i in a minimal relaxed solution $P = \langle o_1, \ldots, o_n \rangle$ there is a legal generation path P_i starting with o_i.

Proof Sketch: (**to Theorem 2**) The desired paths P_i can be constructed by starting with o_i, successively stepping onto a successor action that has a needed fact as precondition, and stopping when a goal fact is needed. With Lemma 2, there is always at least one needed fact, and with Lemma 1, those facts are added. The resulting action sequence is obviously a generation path, and it is legal because facts are not yet true at the position where they are needed. ■

Unfortunately, deciding about the sufficient condition given by Corollary 1 is still NP-hard.

Definition 6. *Let LEGAL-GENERATION-PATH denote the following problem:*
Given a planning task $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ and an action $o \in \mathcal{O}$, is there a legal generation path starting with o?

Theorem 3. *Deciding LEGAL-GENERATION-PATH is NP-complete.*

Proof Sketch: Membership follows by a simple guess-and-check argument. Hardness can be proven by a polynomial reduction from 3SAT. Introduce one action for each literal in the clauses, and one action for each variable. Additionally, introduce a starting action s. The preconditions and add lists can be arranged such that the following holds: Firstly, a generation path starting with s must visit all clauses at least once, and afterwards pass through all variables. Secondly, passing a variable legally requires that the path has not visited the respective variable *and* its negation. A legal generation path starting in s thus defines a satisfying truth assignment via the literals visited in the clauses, and vice versa. ■

4 Approximation Techniques

We will now introduce two polynomial-time approximations of legal generation paths, filtering action sets for relaxed planning. The first method includes all actions that start a legal path, and is therefore complete in the relaxation. As we will see in the next section, the method terminates fast in almost all of our testing examples. The second method does not give any completeness guarantees, but will be shown to be empirically safe, and to terminate extremely fast on *all* examples in our testing suite.

4.1 A Sufficient Approximation

Let us first introduce a notation for the set of all actions that start a legal generation path. With Corollary 1, we can restrict the actions considered by an FF or HSP style heuristic function to that set without loosing completeness.

Definition 7. *Let $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task. The* legal action set *to $\mathcal{P}$ is $\mathcal{O}|_l := \{o \in \mathcal{O} \mid \exists\, P \in \mathcal{O}^* : \langle o \rangle \circ P$ is a legal generation path $\}$.*

Our sufficient approximation collects together all actions starting generation paths that fulfill a weaker notion of legality. Reconsider Definition 4.

Definition 8. *Let $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task. A generation path $P = \langle o_1, \ldots, o_n \rangle$ is* initially legal *if $(add(o_i) \cap pre(o_{i+1})) \setminus pre(o_1) \neq \emptyset$ for $1 \leq i \leq n-1$. The* initially legal action set *$\mathcal{O}|_{il}$ to $\mathcal{P}$ is defined using the following fixpoint operator $\Gamma : 2^{\mathcal{O}} \mapsto 2^{\mathcal{O}}$.*

$$\Gamma(\mathcal{O}|_r) := \{o \in \mathcal{O} \mid \exists\, P \in \mathcal{O}|_r^* : \langle o \rangle \circ P \text{ is an initially legal generation path } \}$$

We set $\mathcal{O}|_{il} := \bigcup_{i=0}^{\infty} \Gamma^i(\emptyset)$.

In words, we obtain the initially legal action set by computing a fixpoint over the actions that start an initially legal generation path. A generation path is initially legal when between any two actions there is a connecting fact that is not contained in the precondition of the first action. Clearly, legal generation paths—where there are connecting facts that are not contained in the precondition of *any* previous action—fulfill this property.

Proposition 1. *Let $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning task. The initially legal action set is a superset of the legal action set, i.e., $\mathcal{O}|_{il} \supseteq \mathcal{O}|_l$ holds.*

The definition of $\mathcal{O}|_{il}$ translates directly into the fixpoint computation depicted in Figure 1. Our implementation is straightforward. In each iteration of the fixpoint process, check for all not yet selected actions o whether there is a path to the goals, using only edges that are not excluded by o's preconditions.

We have also implemented two other sufficient approximations of $\mathcal{O}|_l$. One of those weakens $\mathcal{O}|_{il}$ by dropping the condition that the action sequences P must consist of $\mathcal{O}|_{il}$ members. The other method strengthens $\mathcal{O}|_{il}$ by incrementally building a graph of edges that start already selected paths. The required action sequences P must then traverse only edges that are in the graph already. In our experiments, both methods showed significantly worse runtime behaviour than the above $\mathcal{O}|_{il}$ computation. The filtered action sets were, however, the same for all three methods in most of the cases. We therefore chose to concentrate on $\mathcal{O}|_{il}$ as a sufficient approximation.

```
O|_il := ∅
repeat
    Fixpoint := TRUE
    for o ∈ O \ O|_il do
        if there is an initially legal path from o to o_G
                consisting out of actions in O|_il then
            O|_il := O|_il ∪ {o}
            Fixpoint := FALSE
        endif
    endfor
until Fixpoint
```

Fig. 1. Fixpoint computation of actions starting initially legal generation paths: A sufficient approximation of legal generation paths.

4.2 An Insufficient but Fast Approximation

Like the computation of initially legal paths, our second approximation technique performs a fixpoint computation. Unlike the former computation, the method allows only edges (o, o') in the paths that are legal with respect to o. What's more, each action o is associated with at most one single edge that can be traversed from o. We call the resulting action set the set of *approximative legal* actions $\mathcal{O}|_{al}$. Have a look at the pseudo code in Figure 2.

```
O|_al := {o_G}, e := ∅, k := 0
repeat
    Fixpoint := TRUE
    for o ∈ O \ O|_al do
        if there is an edge (o, o'), o' ∈ O|_al such that
                the path ⟨o, e^0(o'), e^1(o'), ..., e^k(o') = o_G⟩ is initially legal then
            O|_al := O|_al ∪ {o}
            e := e ∪ {(o, o')}
            Fixpoint := FALSE
        endif
    endfor
    k := k + 1
until Fixpoint
```

Fig. 2. Fixpoint computation of actions starting approximative legal generation paths: An insufficient but fast approximation of legal generation paths.

The algorithm depicted in Figure 2 iteratively includes new actions into $\mathcal{O}|_{al}$ until a fixpoint is reached. The key feature of the algorithm is the function $e : \mathcal{O} \mapsto \mathcal{O}$, which is represented in the figure as a set of $(o, e(o))$ pairs. The function starts as the empty set of such pairs, i.e., e is initially undefined for the whole action set. If an action o is included into $\mathcal{O}|_{al}$ due to an edge (o, o'), then that edge is included into the definition of e. Initially, the only member of $\mathcal{O}|_{al}$ is o_G, so in iteration $k = 0$ the only edges that can be included are direct connections to the goals. In any later iteration k, e defines a tree of depth k where the root node is o_G, and each node—the actions for which e is defined—occurs exactly once. For the not yet selected actions o it is then checked whether they have an edge connecting them to a tree node o' such that the path $\langle o, e^0(o'), e^1(o'), \ldots, e^k(o') = o_G\rangle$ is initially legal. Note here that $\langle o, e^0(o'), e^1(o'), \ldots, e^k(o')\rangle$ is just the concatenation of the

edge (o, o') with the path from o' to the tree root. If that path is initially legal, then o and the edge (o, o') are included into the tree.[3] While allowing only a single edge for each node may sound way to restrictive, the method turned out to be, as said, surprisingly safe in our testing examples.

5 Extension to Conditional Effects

We have extended our theoretical analysis and approximation algorithms to deal with conditional effects. Because FF compiles away all ADL constructs except the conditional effects [4], this enabled us to deal with planning domains specified in the ADL language [9]. In the following, we briefly summarise the extensions made to the definitions and algorithms introduced in Sections 3 and 4. For more details, we refer the interested reader to our technical report [5].

An effect is relaxed irrelevant if it can be ignored in all optimal relaxed solutions from all reachable states, i.e., if all optimal relaxed plans are still relaxed plans without that effect. Relaxed irrelevant effects can be detected by looking at the set of all effects in a task as a set of STRIPS actions STRIPS($\mathcal{O}$), where each effect of an action o corresponds to an action that has as preconditions pre(o) plus the effect's conditions. The parallel to Theorem 2 is that, if an effect can not be ignored in a minimal relaxed solution from some state, then the effect starts a legal generation path in STRIPS($\mathcal{O}$). This can be proven by a natural extension of the needed facts notion.

Extending the filtering methods from Section 4 thus comes down to implementing them on the set STRIPS($\mathcal{O}$). If an effect does not start an initially or approximative legal generation path in STRIPS($\mathcal{O}$), then the effect is removed from the respective action in the sense that the effect is not considered within the relaxation. If all effects of an action are removed, then the whole action is ignored.

6 Empirical Evaluation

We evaluated our approach by running a number of large scale experiments. We used 20 benchmark planning domains, including all examples from the AIPS-1998 and AIPS-2000 competitions. The domains were *Assembly*, two *Blocksworld*s (three- and four-operator representation), *Briefcaseworld*, *Bulldozer*, *Freecell*, *Fridge*, *Grid*, *Gripper*, *Hanoi*, *Logistics*, *Miconic*-ADL, *Miconic*-SIMPLE, *Miconic*-STRIPS, *Movie*, *Mprime*, *Mystery*, *Schedule*, *Tsp*, and *Tyreworld*. In each of these domains, we generated instances by using randomised generation software.[4] We ran experiments for evaluating

1. RIFO's runtime behaviour when compared to FF,
2. the runtime behaviour and pruning impact of $\mathcal{O}|_{il}$ and $\mathcal{O}|_{al}$,
3. and the empirical safety of $\mathcal{O}|_{al}$.

For each single experiment, we set up a large testing suite containing up to 200 instances from each domain. The testing suites differed in terms of the size of the instances that we generated.

In the first experiment, we ran the RIFO implementation within IPP4.0 versus FF on a suite of 681 instances that were small enough for the IPP4.0 instantiation

[3] For optimisation, one obviously only needs to look at actions o' that are leafs of the current tree.

[4] Descriptions of the randomisation strategies and the source code of all generators are publicly available at http://www.informatik.uni-freiburg.de/~ hoffmann/ff-domains.html.

routine to cope with.[5] Test runs were given 300 seconds time and 400 M Bytes memory on a Sun machine running at 163 MHz. We show the number of instances handled successfully, and the average running time per domain. For FF, we count as successfully handled those instances were a plan was found. For RIFO, success on an instance means termination of the pre-process within the given time and memory bounds. We count only those such instances for which we know they are solvable—those were FF found a plan. Times are averaged over those instances that both implementations handled successfully. Running time for RIFO does *not* include IPP's instantiation time. See the data in Figure 3.

	success		running time	
domain	RIFO	FF	RIFO	FF
Assembly	33	33	1.08	9.16
Blocksworld-3ops	21	21	4.45	2.90
Blocksworld-4ops	21	21	0.91	0.07
Briefcaseworld	20	20	1.86	1.12
Bulldozer	17	17	1.97	4.54
Freecell	33	50	21.90	0.06
Fridge	22	22	0.23	0.22
Grid	22	35	43.77	7.72
Gripper	25	25	0.45	0.31
Hanoi	8	8	0.34	4.79
Logistics	35	35	46.80	1.18
Miconic-ADL	22	40	14.03	3.77
Miconic-SIMPLE	25	25	0.64	0.54
Miconic-STRIPS	25	25	0.64	0.37
Movie	30	30	0.00	0.00
Mprime	48	61	16.47	1.19
Mystery	23	36	27.16	12.51
Schedule	15	28	28.34	14.06
Tsp	25	25	4.90	0.12
Tyreworld	20	20	6.03	0.48

Fig. 3. Instances handled successfully, and average running times for RIFO and FF per domain. The successfully handled instances for FF are those for which a plan was found. The successfully handled instances for RIFO are those solvable ones where RIFO terminated within the given time and memory bounds.

In 3 of the 20 domains shown (*Assembly*, *Bulldozer* and *Hanoi*) does RIFO terminate faster than FF solves the tasks. In 10 domains, RIFO's average running time is orders of magnitude higher than that of FF. In some domains, RIFO exhausts resources on a number of instances that FF manages to solve. We conclude that RIFO is, as a pre-process, not competitive with FF, at least in its implementation within IPP4.0.

In our second experiment, we evaluated the $\mathcal{O}|_{il}$ and $\mathcal{O}|_{al}$ methods in terms of runtime behaviour and pruning impact. Test runs were given 300 seconds and 200 M Bytes memory on a Sun machine running at 300 MHz. We used a total of 2334 large instances generated to be of a size challenging for FF, but still within its range of solvability within the given resources. On each task, we ran three implementations: FF-v2.2 [4], and two versions of the same code were $\mathcal{O}|_{il}$ respectively $\mathcal{O}|_{al}$ were computed as a pre-process. In the latter two versions, FF's heuristic function was changed to consider only those effects contained in the filtered action set. We measured the overhead produced by the filtering methods, the total running times, the time taken for state evaluations, and the number of effects in the complete respectively filtered action sets. See the data in Figure 4.

[5] In some domains, like *Freecell*, the routine can handle only comparatively small instances which is, we think, due to the implementation: this is intended to deal with full scale ADL constructs [6], and fails to efficiently handle the simple STRIPS special case.

domain	overhead $\mathcal{O}\|_{il}$	overhead $\mathcal{O}\|_{al}$	total time FF	total time $+\mathcal{O}\|_{il}$	total time $+\mathcal{O}\|_{al}$	single evaluation FF	single evaluation $+\mathcal{O}\|_{il}$	single evaluation $+\mathcal{O}\|_{al}$	number of effects $\mathcal{O}$	number of effects $\mathcal{O}\|_{il}$	number of effects $\mathcal{O}\|_{al}$
Assembly	0.01	0.00	12.83	12.01	11.53	1.75	1.64	1.57	426.72	358.64	358.64
Blocksworld-3ops	0.59	0.08	1.62	2.24	1.61	3.41	3.47	3.21	1854.62	1854.62	1819.09
Blocksworld-4ops	0.04	0.00	1.04	1.08	0.99	0.76	0.76	0.72	290.06	290.06	286.94
Briefcaseworld	0.04	0.01	5.51	1.10	1.01	4.26	0.82	0.77	4106.50	670.00	670.00
Bulldozer	0.02	0.01	6.89	7.00	6.67	1.27	1.29	1.23	599.22	599.22	599.17
Freecell	11.14	0.53	17.47	28.77	17.33	8.19	8.26	7.87	4725.37	4668.17	4668.17
Fridge	0.00	0.00	1.71	1.72	1.70	0.96	0.97	0.95	302.22	302.22	302.22
Grid	76.12	0.54	11.57	87.90	11.93	7.95	8.09	7.89	6424.35	6424.35	6417.28
Gripper	0.03	0.01	0.33	0.36	0.27	1.38	1.39	1.11	478.00	478.00	359.00
Hanoi	0.00	0.00	4.73	4.80	4.58	0.83	0.84	0.80	244.50	244.50	244.50
Logistics	2.24	0.22	83.57	45.51	43.52	37.45	19.39	19.40	19904.53	15347.80	15347.80
Miconic-ADL	1.09	0.29	13.91	13.48	12.23	12.72	11.33	10.92	2988.20	2700.52	2700.52
Miconic-SIMPLE	0.17	0.02	0.52	0.69	0.51	2.14	2.15	2.04	1504.00	1504.00	1504.00
Miconic-STRIPS	0.16	0.02	0.39	0.55	0.38	1.92	1.94	1.82	1504.00	1504.00	1504.00
Movie	0.00	0.00	0.00	0.00	0.00	0.33	0.23	0.17	7.00	7.00	7.00
Mprime	60.20	0.79	5.40	65.66	6.13	16.38	16.54	16.18	12138.00	12136.97	12136.32
Mystery	12.87	1.05	20.11	33.26	21.14	15.59	15.80	15.57	14644.20	14644.20	14641.38
Schedule	0.48	0.01	52.31	55.56	54.52	10.86	7.07	6.99	3049.84	917.43	916.82
Tsp	0.01	0.09	0.13	0.14	0.22	2.09	2.11	2.13	4390.00	4390.00	4390.00
Tyreworld	1.34	0.08	23.23	13.27	7.07	19.31	9.92	5.81	7105.50	4479.00	3646.00

Fig. 4. Average overhead for pre-processing, average total running time, average running time per state evaluation, and average number of effects, shown per planning domain and filtering method used. Times are in seconds except for state evaluations, where milliseconds are specified.

All measured values were averaged over those instances were all three methods succeeded in finding a plan (we tried inserting default values in the other cases, but found that this generally obfuscated the results more than it helped understanding them). In 12 domains, the solved instances were exactly the same across all methods anyway. In another 3 domains, differences occurred only in very few instances (1 - 2 out of 90 - 181). In *Grid* and *Mprime*, computing $\mathcal{O}|_{il}$ sometimes exhausted resources (in *Grid*, 41 of 179 cases, in *Mprime*, 51 of 196 cases). In *Assembly* and *Logistics*, the speed-up produced by the filtering methods helped FF to solve some more instances (165 instead of 159 in *Assembly*, 87 instead of 75 in *Logistics*). In *Schedule*, original FF solved 85 instances instead of 74 solved with $\mathcal{O}|_{il}$ or $\mathcal{O}|_{al}$ on. We will come back to the *Schedule* domain later.

Let us first focus on the overheads produced. Compare the first two columns with the third column, showing average solving time for FF. The overhead for $\mathcal{O}|_{il}$ is neglectible (i.e., below 0.2 seconds on average) in 11 of our domains, and orders of magnitude smaller than FF's average time in another 4 domains. In the 3-operator *Blocksworld*, the overhead is a third of *FF*'s time, and below a second anyway. In the remaining four domains, the pre-process can hurt: In *Freecell* and *Mystery*, it takes almost as much time as FF, and in *Grid* and *Mprime* it can take much longer time (we will later describe an approach to automatic recognition of the cases were the pre-process takes a lot of time). The overhead for $\mathcal{O}|_{al}$ is neglectible in 14 of the domains, and still a lot smaller than FF's running time in the other cases.

Concerning the impact that the filtering methods have on the number of effects in the action set, the speed of the heuristic function, and the total running time, it is easiest to start by looking at the rightmost three columns in Figure 4. The methods do not prune any effects in 6 of our domains, and prune very few effects in another 7 domains. Moderately many effects are pruned in the *Assembly*, *Gripper* and *Miconic*-ADL domains. In the *Briefcaseworld*, *Logistics*, *Schedule* and *Tyreworld* domains, the pruning is drastic.[6] As a consequence, the average time taken

[6] In the *Briefcaseworld*, for example, amongst other things all actions are thrown out that take objects out of the briefcase—taking objects out of the briefcase is not necessary within the relaxation, where keeping them inside never hurts.

for a single state evaluation (total evaluation time divided by number of evaluated states) is, when using the filtering methods, significantly lower in the four domains with drastic pruning, and slightly lower in the three domains with moderate pruning. Look at the respective columns, specifying the average state evaluation time in milliseconds. In *Briefcaseworld*, *Logistics* and *Tyreworld*, the faster heuristic functions translate directly into improved total running time. In *Schedule*, there seems to be some interaction between the filtering methods and FF's internal algorithmic techniques: though the heuristic function is faster, total running time gets worse. This is because FF evaluates, with the filtered action sets, more states before finding the goal. An explanation for this might be FF's helpful actions heuristic, which biases the actions selected to those that could also be selected by the heuristic function [4]. For $\mathcal{O}|_{al}$, it might also be that some states become unsolvable—though we did not find such a case in the experiment described below.

We finally consider the safety of the $\mathcal{O}|_{al}$ filtering method with respect to completeness in the relaxation. The method is empirically safe in the sense that, from the 2334 examples used in the above described experiment, only 11 *Schedule* instances could not be solved with the method on though they could be solved with original FF. The failures were only due to the runtime restrictions we applied in the experiment: given slightly more time, FF with $\mathcal{O}|_{al}$ filtering *could* solve those 11 instances. In addition to this result, we ran the following experiment. We generated a total of 2099 instances from our 20 domains, small enough to build an explicit representation of the state space. To each instance, we looked at all reachable states, and verified whether the goal was reachable when ignoring delete lists, using the whole action set $\mathcal{O}$, or the filtered action set $\mathcal{O}|_{al}$. In 19 of our 20 domains, all states solvable with $\mathcal{O}$ were still solvable with $\mathcal{O}|_{al}$. Only in *Grid* did we find states that became unsolvable. This occurred in 19 of 100 instances. In all those instances, the states becoming unsolvable were less than 1% of the state space.

7 Current Work

Our current results reveal two drawbacks of the presented approach:

1. $\mathcal{O}|_{il}$ filtering sometimes hurts in the sense that it can take a lot of running time.
2. While $\mathcal{O}|_{il}$ is provably and $\mathcal{O}|_{al}$ empirically safe, both methods have strong pruning impacts only in a few domains.

We address these difficulties in two lines of work that we are currently pursuing. One idea to avoid the first problem is estimate the runtime that would be necessary for computing $\mathcal{O}|_{il}$. One can then skip the pre-process if it appears to be too costly. $\mathcal{O}|_{il}$ is computed by the repeated search for legal generation paths, which is more costly the more edges there are in the generation graph. An upper approximation to the number of edges is:

$$\sum_{f \in F} |\{o \in \mathcal{O} \mid f \in \mathrm{add}(o)\}| * |\{o \in \mathcal{O} \mid f \in \mathrm{pre}(o)\}|$$

Here, F denotes the set of all logical atoms that appear in the actions $\mathcal{O}$. If $|\mathrm{pre}(o) \cap \mathrm{add}(o')| \leq 1$ for all $o, o' \in \mathcal{O}$, then the approximation is exact. We have computed, for the 2334 large instances from the second experiment described in the previous section, the above upper limit, as well as the real number of edges in the generation graph. In 8 domains, the values are the same across all instances. In the remaining domains, the values are close. There seems to be a close correspondence to the running time consumed by the $\mathcal{O}|_{il}$ computation: the averaged approximation values are between 3 and 11 millions in those four domains were $\mathcal{O}|_{il}$ takes a lot of

computation time, and below one million in all other domains. It remains to establish an exact criterion that uses this correspondence for deciding about whether to compute $\mathcal{O}|_{il}$ or not.

Addressing the second problem, lack of strong pruning impacts in many domains, appears to us to be a much harder task. If one wants to obtain stronger pruning impacts, there does not seem to be a way around sacrificing empirical, let alone theoretical safety. We are currently experimenting with combining our techniques and RIFO's information selection heuristics. We have implemented some first strategies. As expected, the pruning impact became more drastic in some examples. However—as we also expected—a lot of states became unsolvable for the heuristic. Often all paths to the goal were interrupted by such a state, rendering the whole planning task unsolvable for FF.

8 Conclusion and Outlook

We have presented a new approach towards defining irrelevance in planning tasks, concerning actions that are not necessary within the relaxation used in the heuristic functions of state-of-the-art heuristic planners like HSP and FF. We have derived a sufficient condition for relaxed irrelevance, and we have presented two approximation methods that can be used for filtering action sets. One of those methods, $\mathcal{O}|_{il}$ computation, has been proven to be complete within the relaxation, the other method, $\mathcal{O}|_{al}$ computation, has been shown to be empirically safe. The methods have drastic pruning impacts in some domains, speeding up FF's heuristic function, and in effect the planning process (except in *Schedule*, where there appears to be some interaction with FF's internal techniques). Computing $\mathcal{O}|_{al}$ never hurts in the sense that the required overhead is neglectible in most of the cases, and always small compared to FF's running time. Computing $\mathcal{O}|_{il}$ does not hurt in 16 of our 20 domains. We have outlined an approach how the other cases might be recognisable automatically. The challenge remains to find filtering methods that are still empirically safe in most of the cases, but have stronger pruning impacts.

References

1. Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):279–298, 1997.
2. Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 2001. Forthcoming.
3. Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.
4. Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
5. Jörg Hoffmann and Bernhard Nebel. RIFO revisited: Detecting relaxed irrelevance. Technical Report 153, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany, 2001. Available from `http://www.informatik.uni-freiburg.de/tr/2001`
6. Jana Koehler and Jörg Hoffmann. On the instantiation of ADL operators involving arbitrary first-order formulas. In *Proc. ECAI-00 Workshop on New Results in Planning, Scheduling and Design*, 2000.
7. Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In *Proc. ECP-97*, pages 273–285, Toulouse, France, September 1997. Springer-Verlag.
8. Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler. Ignoring irrelevant facts and operators in plan generation. In *Proc. ECP-97*, pages 338–350, Toulouse, France, September 1997. Springer-Verlag.
9. Edwin P.D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR-89*, pages 324–331, Toronto, ON, May 1989. Morgan Kaufmann.

Using Reactive Rules to Guide a Forward-Chaining Planner

Murray Shanahan

Department of Electrical and Electronic Engineering,
Imperial College,
Exhibition Road,
London SW7 2BT,
England.
m.shanahan@ic.ac.uk

Keywords: planning and execution, reactive planning, robot planning

Abstract

This paper presents a planning technique in which a flawed set of reactive rules is used to guide a stochastic forward-chaining search. A planner based on this technique is shown to perform well on Blocks World problems. But the attraction of the technique is not only its high performance as a straight planner, but also its anytime capability. Using a more dynamic domain, the performance of a resource-bounded version of the planner is shown to degrade gracefully as computational resources are reduced.

1 Introduction

As Bacchus and Kabanza have demonstrated, the use of domain-specific rules to guide a forward-chaining planner is a promising line of research [Bacchus & Kabanza, 2000]. Using only a handful of temporal logic formulae as heuristics, their TLPlan system outperforms many state-of-the-art domain-independent planners on hard Blocks World problems. In a similar vein, the present paper describes a forward-chaining planner whose search is controlled by domain-specific rules. But where TLPlan uses formulae of temporal logic to guide the search, the present planner uses a set of reactive rules.

Although the resulting planner is reasonably fast, the aim of the present work isn't the construction of a high-performance planner. Rather, the chief aim is to supply a system that seamlessly integrates reactive behaviour and planning at the same level. This is achieved because the reactive rules enable the planner to behave as an anytime algorithm [Dean & Boddy, 1988], [Zilberstein, 1993]. The planner can always supply a partial solution, including an action to execute right away, no matter how far into its computation it has gone. But the solutions it finds increase in quality with the time available to search for them.

Given sufficient time to respond, the planner generates a complete plan from initial state to goal. In this case, the reactive rules serve to speed up the search. Given insufficient time to find a complete plan, the system responds with a partial plan constructed (mostly) out of actions recommended by the reactive rules, but with the benefit of look-ahead to favour those that are least dangerous and most promising. Given very little time to respond, the system doesn't even have the luxury of look-ahead, and all it can offer is a partial plan comprising a single action recommended by the reactive rules. In other words, it starts to behave as the reactive rules would on their own.

Although some pioneers of the use of reactive rules rejected deliberative planning altogether [Brooks, 1986], [Agre & Chapman, 1987], in the late Eighties and early Nineties, a variety of ways of reconciling planning with reactivity were studied. Prominent examples include the work of Schoppers [1987], Drummond [1989], and Mitchell [1990]. In robotics, hybrid architectures combining a low-level reactive layer with a high-level deliberative layer, such as that described by Gat [1992], have become commonplace, and are now to be found in the remotest corners of the Solar System [Pell, *et al.*, 1997]. However, none of this work looks at the possibility of using reactive rules to guide a forward-chaining planner, possibly because the idea only starts to look plausible in the light of Bacchus and Kabanza's more recent work.

2 Useful but Imperfect Reactive Rules

Consider the following pair of goal-directed reactive rules for solving Blocks World problems. As we'll see, these rules are useful but imperfect.

RULE BW1

Move X onto Y if X is on Y in the goal and everything under Y is correct with respect to the goal

RULE BW2

Move X from Y to Z if anything under Y is incorrect with respect to the goal and everything under Z is correct with respect to the goal.

Let's investigate the performance of these rules on some examples, beginning with the trivial Blocks World problem depicted in Figure 1. Starting in the initial state, the successive application of Rules BW1 and BW2 yields the following sequence of actions leading to the goal: move A to the table, move B to the table, move C to B, move A to C.

Figure 1: Blocks World Problem `bw-small`

In this case, the rules generate a unique sequence of actions. But in general, such rules are non-deterministic. For example, in the initial state of the Blocks World problem in Figure 2, the rules recommend three possible actions – move E to the table, move C to the table, and move I to the table.

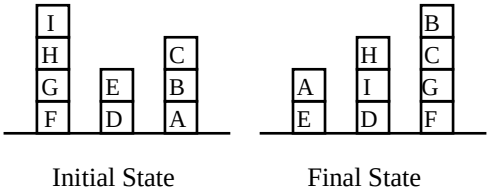


Figure 2: Blocks World Problem `bw-large-b`

Rules BW1 and BW2, though simple, are effective with many Blocks World problems. Rule BW1 alone can generate an optimal 6-step solution to the problem of Figure 2 using negligible CPU time. However, these two rules aren't guaranteed to find a solution to *any* Blocks World problem. Problem `bw-large-b+` is identical to `bw-large-b` except that blocks E and A are swapped in the final state. Given this problem, one possible sequence of applications of Rules BW1 and BW2 leads to the state shown in Figure 3. In this state neither Rule BW1 nor Rule BW2 is applicable.

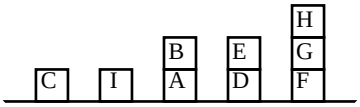


Figure 3: A Stalled State

Not only can the rules lead to stalled states. They can also recommend actions that are positively harmful. Take the problem in Figure 4, for example. In the initial state, one of the actions recommended by the reactive rules is to move A onto C, where it will prevent B from getting to its final destination.

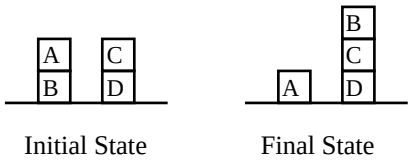


Figure 4: Rule BW2 Goes Wrong

In general, the fact that a set of reactive rules has been effective on a large class of problems is no guarantee that it will solve the next problem that comes along. If the rules are hand-coded or pre-computed off-line, their correctness and completeness can be proved. But if the rules are learned on the fly, by a reinforcement learning algorithm, for example, there's no guarantee of completeness, and no possibility of off-line validation.

It's easy to see that, in this case, an additional rule would solve the stalled state problem. The new rule would cater for the case when X needs to be moved off Y, even though the blocks below Y are correctly placed. But this is beside the point. The pertinent observation here is that an imperfect set of rules, of the sort that a learning algorithm might generate, can still be useful.

The system proposed here is designed to exploit the availability of useful but imperfect sets of reactive rules. As long as the rules are effective some of the time, they can be used to efficiently guide the search of a forward-chaining planner. The reactive rules serve a similar function to the temporal logic formulae in TLPlan. But unlike TLPlan's heuristic formulae, the reactive rules can be used in stand-alone mode when a fast response is needed. Moreover, although the examples of reactive rules used in this paper are all indexed on a goal, the system can also function with goal-less reactive rules (of the kind that are prevalent in biologically-inspired approaches to robotics). In the absence of an explicit goal, it isn't an advisable strategy for an intelligent agent to cease activity altogether, which is why goal-less reactive rules that encourage exploration and play are useful. Furthermore, at times when there is no explicit goal, an agent still has to react to ongoing events. In either case, it's advantageous for the agent

to look ahead before executing an action. In contrast to conventional approaches to planning and plan execution, the present system can accommodate this requirement.

3 Linear Iterative Lengthening

Given unbounded time to act, the system under discussion is guaranteed to produce a plan if one exists. Given very little time to act, the system can still suggest the next action to be executed using the reactive rules alone. In between these two extremes, the reactive rules and the search-based planner operate together, producing a high-quality partial plan with an immediately executable action. Moreover, as we'll see, in this intermediate region, the system can discover plans that the reactive rules would miss altogether and that a conventional search-based planner would fail to find within an acceptable time.

The basic idea of using reactive rules to guide a forward-chaining planner can be implemented in a variety of ways. During the work carried out for this paper, several different approaches were tried, on a variety of Blocks World problems, using Rules BW1 and BW2 from Section 2 as the reactive guide. Only the most promising technique is described here. This also happens to be the simplest of the implementations.

The most successful planner uses an algorithm we'll call *linear iterative lengthening*. In the following pseudo-code, the algorithm is invoked by a call to ILPlan(G,S,P), where G is the goal state, S is the initial state, and P is the returned plan. A state is a set of fluents, to which the closed world assumption is applicable. The variable L is the length bound.

```
1   Procedure ILPlan(G,S,P)
2       L := 1
3       FPlan(G,S,L,P)
4       While plan not found
5           L := L + 1
6           FPlan(G,S,L,P)
```

The FPlan procedure tries to construct a plan, and exits if it finds one or if the plan it's working on exceeds the current length bound. In this case, the length bound is incremented, and the planner goes round the loop again, starting with an empty plan.

```
7   Procedure FPlan(G,S,L,P)
8       P := []
9       While Length(P) < L and S ≠ G
10          Let RL be the set of actions recommended
11              by the reactive rules for state S
12          Let EL be the set of remaining actions
13              executable in S
14          If RL is empty
15          Then randomly select A from EL
16          Else randomly select A from either
17              EL or RL with a bias towards RL
18          P := [A | P]
19          S := result of executing A in S
```

The FPlan procedure uses forward chaining to construct a single candidate plan in progression order. Successive actions are randomly selected, but with a heavy bias towards those recommended by the reactive rules.

While this planner is sound, it obviously isn't complete. No attempt is made to systematically explore the search tree. It is, however, complete in a stochastic sense: as the execution time tends to infinity, the probability of finding a solution, if one exists, tends to one. (In the reported experiments, though, the bias towards recommended actions was made 100%, sacrificing completeness even in this stochastic sense.) In practise, theoretical completeness is a less interesting property than how likely it is that the planner will actually find a solution in an acceptable timeframe. On this score, how good can an algorithm be that burrows apparently blindly down a single branch of the search tree at a time?

The answer is that everything depends on the quality of the reactive rules guiding the planner. At one extreme, if the reactive rules are very poor, the planner really is working blindly and its performance is appalling. At the other extreme, if the reactive rules are extremely good, they can solve any problem directly, and the planner is redundant. The planner's performance is most impressive in the intermediate cases, where the reactive rules are effective on a small class of problems, but are unable to solve all problems by themselves. As we'll see in the next section, the Blocks World rules of Section 2 fall into this intermediate category, and the planner's performance is correspondingly impressive.

But for the planner to be efficient, even in these intermediate cases, it has to be able to make effective guesses when the reactive rules are inapplicable. The algorithm above simply makes a random choice. This works fine with the Blocks World, because it's just enough to jog the planner out of a stalled state. But in other domains, it will be necessary to employ a little more sophistication. One way to do this is to introduce an evaluation function for scoring potential successor states, and to use this to select the most promising (and least dangerous) next action. This extension to the basic algorithm will be discussed in Section 6.

4 Blocks World Experiments

As already stated, the thrust of this paper is *not* the design of a stand-alone high-speed planner. The issue at hand is architecture rather than performance. In designing a whole agent, such as a robot, we need to integrate features such as deliberative planning, reactive behaviour, and heuristics in a single control system that also interleaves computation and action. Accordingly, the experimental results presented in this section should be taken solely as a "proof of concept".

The result of applying the planner to a number of Blocks World problems are presented, using the two reactive rules discussed in Section 2 to guide the search. Problems `bw-small`, `bw-large-b`, and `bw-large-b+` were described in Section 2. Problems `bw-large-c` and `bw-large-d` were taken from the BlackBox test suite [Kautz & Selman, 1999]. Problem `bw-large-c+` is a modification of `bw-large-c`. Each problem was submitted to the planner 10 times, first using only Rule BW1, and then using both Rules BW1 and BW2. The results using only Rule BW1 are presented in Table 1.

The planner was implemented in LPA Prolog 32 on an Apple iMac with a 233MHz G3 processor. Note that, in a sense, the true metric here is not the time taken to find a

solution, but whether or not the planner can find an answer *at all* within a reasonable timeframe. If a planner can deliver an answer to a problem in a matter of seconds, then turning this into milliseconds is simply a matter of optimisation and computing power. On the other hand, when the system fails to present an answer having been left to run overnight, we suspect its practical limits have been reached.

Problem	Av. Time	Av. Length
bw-small	0.02s	4.0
bw-large-b	0.12s	6.0
bw-large-b+	0.19s	7.0
bw-large-c	1.60s	14.9
bw-large-c+	7.13s	32.9
bw-large-d	33.88s	18.5

Table 1: Using Rule BW1 Only

The planner consistently solves `bw-large-b` in around one tenth of a second. The plan is always 6 steps long, which is optimal. Recall that the reactive rules alone fail to solve this problem altogether. Moreover, 9-block problems such as `bw-large-b` and `bw-large-b+`, though soluble in seconds by the current generation of domain independent planners, exemplified by BlackBox [Kautz & Selman, 1999], were large enough to overwhelm earlier planners, such as UCPOP [Penberthy & Weld, 1992].

The present planner consistently solves the 15-block problem `bw-large-c` in less than 2 seconds. However, `bw-large-c` is a relatively "easy" problem for its size. A solution can be found just by repeatedly following Rule BW1. In other words, there's always a block that can be moved to its final destination, and the planner never has to dismantle a tower just to get at a particular block.

Problem `bw-large-c+` is designed to be more difficult. The initial state is that of `bw-large-c`. But the goal state is the same as the initial state with only the bottom two blocks of each tower swapped over. This requires the planner to take each tower apart, swap the bottom blocks, and then rebuild it. The planner also performed satisfactorily on this problem, consistently finding a solution in under 10 seconds.

Finally, the planner was run on a 19-block problem, `bw-large-d`. Until recently, Blocks World problems of this size were beyond the capability of domain-independent planners. On average, the present planner can solve `bw-large-d` in around 30 seconds.

Problem	Av. Time	Av. Length
bw-large-a	0.02s	4.0
bw-large-b	0.72s	8.5
bw-large-b+	0.96s	9.5
bw-large-c	14.27s	20.0
bw-large-c+	21.52s	24.0
bw-large-d	58.36s	28.4

Table 2: Using Rules BW1 and BW2

Table 2 presents the results of running the planner on the same problems, but using both Rules BW1 and BW2. The performance turns out to be worse than with Rule BW2

absent, with respect to both time and length. This is because of Rule BW2's tendency to recommend the occasional harmful action. Since the recommendations of reactive rules are given such weight, their mistakes are very costly.

5 The Kids World Domain

The results reported in the previous show that the planner outperforms fast domain-independent planners in the Blocks World, using only a single guiding reactive rule, one that is incapable of solving Blocks World problems on its own. This is an encouraging result. But it doesn't show off one of the most attractive features of a forward-chaining planner guided by reactive rules, which is its anytime capability.

In this section, a new domain is introduced, called Kids World, which is adjustably dynamic, in the sense that unexpected events can occur with a preset probability. (In other respects, Kids World resembles the Logistics domain used to assess planners in planning competitions.) Kids World problems will be tackled with a resource-bounded version of the planner of Section 3, which is embedded in a sense-plan-act cycle that executes plans and reacts to ongoing events.

Two properties of this system are demonstrated using Kids World problems. First, it's demonstrated that the planner's performance degrades gracefully as its resource bound is decreased. Second, it's shown that the system can solve Kids World problems in real time, even with the frequent occurrence of unexpected events.

The Kids World domain comprises the four fluents and five actions summarised in Table 3 below. The action Move(x) affects the location of both the parent and any child they are carrying. The Open(d) and Close(d) actions have the precondition that the parent isn't carrying anything. To move from one location to another, the connecting door has to be open.

The particular Kids World problem used in the experiments reported here involves two children, Kerry and Liam, and three locations — the house, the street, and the car. These locations are connected by doors in the obvious way. In the initial state, the parent and both children are in the house and the doors are all closed. In the final state, everyone is in the car and both the children are happy. A crucial additional twist to the problem is that Kerry becomes unhappy if Liam is put in the car first.

Fluent/Action	Meaning
Location(x)	Parent is in location x
Location(c,x)	Child c is in location x
Carrying(c)	Parent is carrying child c
Happy(c)	Child c is happy
IsOpen(d)	Door d is open
Move(x)	Go to location x
PickUp(c)	Pick child c up
PutDown(c)	Put child c down
Open(d)	Open door d
Close(d)	Close door d

Table 3: Kids World Fluents and Actions

Planning in Kids World would be much easier if children stayed where they were put. Then, planning could be studied without having to consider plan execution. But to make

Kids World problems realistic, after every action the parent performs, there's a certain probability that one or other of the children will run off somewhere completely unexpected. In Experiments `kw-1` and `kw-2` described in the next section, this probability is set to zero, but in Experiment `kw-3`, it is set to 0.1.

A set of seven reactive rules were used in the experiments reported here. Two examples follow. Note their non-determinism. Suppose the parent is in the car carrying Liam, while Kerry is still in the street. Then Rule KW1 recommends putting Liam down, while Rule KW2 recommends going back for Kerry.

RULE KW1

If you're carrying a child and you're in the location the child has to end up, then put the child down

RULE KW2

If there's a child in another room who needs to be moved, then move towards that room

As with the Blocks World example, these seven reactive rules are deliberately imperfect. On their own, they can solve some Kids World problems. But they cannot solve the problem described above on their own, as they quickly lead to a stalled state. Even with the injection of the occasional random action to jog the system out of a stalled state, the rules still frequently make the fatal mistake of putting Liam in the car first. However, the rules can be used to effectively guide a forward chaining planner.

6 Planning with Bounded Computation

This section describes the resource-bounded version of the planner. With a low resource bound, the system amounts to the use of reactive rules with look-ahead. But with higher values, the system doesn't commit to an action early by executing it as reactive rules do, even with look-ahead. Instead, it carries out search — backtracking "in the head" rather than in the world.

The linear iterative lengthening algorithm of Section 3 can easily be made resource-bounded by incorporating a count of the number of times the body of the while loop in the FPlan procedure is executed. The ILPlanR procedure below maintains a resource bound V. The planning process is terminated if V exceeds a predefined bound Φ.

```
1  Procedure ILPlanR(G,S,P)
2      L := 1; V := 0
3      FPlanR(G,S,L,P,V)
4      While plan not found and V < Φ
5          L := L + 1
6          FPlanR(G,S,L,P,V)
```

If V exceeds Φ, the system returns the best partial plan it has found so far. The best partial plan so far is maintained as B in the FPlanR procedure below, and is determined using an evaluation function, Score, on partial plans (see [Korf, 1990]).

```
7   Procedure FPlanR(G,S,L,P,V)
8       P := []; B := []
9       While Length(P) < L and S ≠ G and V < Φ
10          Let RL be the set of actions recommended
11              by the reactive rules for state S
12          Let EL be the set of remaining actions
13              executable in S
14          If RL is empty
15          Then randomly select A from EL
16          Else randomly select A from RL
17          P := [A | P]
18          S := result of executing A in S
19          If Score(P) ≤ Score(B) Then B := P
20          V := V + 1
21      If V ≥ Φ Then P := B
```

When this planner is embedded in a sense-plan-act cycle, it exhibits anytime properties even *without* a carefully designed evaluation function. Table 4 shows the results of Experiment `kw-1`, a Kids World experiment in which Score(P) = 0 for all P. In other words, the planner simply returns the partial plan it's currently working on when the resource bound is exceeded.

In Experiment `kw-1`, the planner has to plan and carry out a sequence of actions that solves the Kids World problem described above. The probability of an unexpected event after a parental action is set to zero. In other words, the kids stay put. After each action the system executes, it *replans from scratch*. This is perfectly feasible, as the planner solves Kids World planning problems very quickly.

Resource Bound	Av. Actions	Aborts
1000	17.4	0
500	17.0	1
200	17.9	5
100	22.5	10
50	23.0	16
10	24.9	15
2	–	30

Table 4: Results of Experiment `kw-1`

The system was run 30 times for each value of the resource bound. A run was aborted after 50 actions, as this indicates that the planner has made Kerry irreversibly unhappy by putting Liam in the car first. The number of actions executed was averaged over all the successful runs. As Table 4 shows, the performance of the system degrades gracefully as the resource bound is reduced, both in terms of the number of aborted runs and the length of the successful runs.

When the resource bound is 1000, the planner can always find a solution, so there are no aborted runs. When the resource bound is 2, the system is behaving almost as if it

were using the reactive rules on their own, and all runs are aborted. For intermediate values of the resource bound, we get intermediate levels of success. When the resource bound falls below 200, we start to see an increase in the length of the successful runs as well as a continuing increase in the number of aborted runs.

To see why this graceful degradation occurs, in spite of the fact that the planner doesn't evaluate intermediate, partial plans, consider how the system behaves as it gets closer to the goal. When the goal is too far away, the planner's resource bound will be exceeded before it finds a plan, and the first action in the partial plan it returns will be one the reactive rules would have chosen on their own. But the closer the goal gets, the more likely it becomes that the planner will find a complete plan before the resource bound is exceeded. The higher the resource bound, the more rapidly this likelihood increases.

Although the planner exhibits anytime properties even when the evaluation function returns a constant, the degradation in the planner's performance can be made more graceful still if a less trivial evaluation function is used. Without such an evaluation function, the partial plans returned when the planner fails to find a complete plan are of uniform quality whatever the resource bound. With a proper evaluation function, a high resource bound means the planner is more likely to find a complete plan, just as before. But it will also produce better quality partial plans when it can't find a complete plan.

In Experiment `kw-2`, a very simple evaluation function was used to demonstrate this. This function simply gives a score of –1 to any partial plan that leaves a child unhappy, and a score of 0 to all other partial plans. The rest of the details of Expriment `kw-2` are as for `kw-1`. The results of this experiment are presented in Table 5, for low values of the resource bound.

Resource Bound	Av. Actions	Aborts
100	20.9	6
50	23.1	11
10	29.0	13
2	32.4	13

Table 5: Results of Experiment `kw-2`

Table 5 shows a clear improvement in performance over that obtained with the constant evaluation function, in terms of aborted runs. Even when the resource bound is 2, the planner is still able to find solutions. The increase in average run length at resource bound 10 is misleading, because it's due simply to the presence of a number of long runs that would have led to abortion with the constant evaluation function. When the resource bound dips below about 50, the number of aborted runs levels off at around 12, although the number of actions per run continues to increase as the resource bound goes down. With such a low resource bound, the planner will rarely find a complete plan, so this gradual degradation can only be put down to a correspondingly gradual reduction in the quality of its partial plans, which is what we were aiming for.

In Experiments `kw-1` and `kw-2`, there were no surprise interventions by the children themselves. But in Experiment `kw-3`, the dynamic potential of the domain is explored, and the probability of a child unexpectedly moving somewhere by itself is increased

from zero to 0.1. Sometimes these unexpected events make the planner's job easier, but most of the time, they make it harder.

Table 6 presents the results of Experiment `kw-3`. The set-up is essentially the same as in Experiment `kw-2`, except that the system is given up to 100 actions before a run is aborted. As before, 30 runs were carried out for each value of the resource bound. In addition to the number of actions averaged over the successful runs, the mean response time per action is given, to indicate that planning is taking place in a realistic timeframe for robotic applications.

The results of Experiment `kw-3` show that the system is consistently able to solve the problem in spite of interference in plan execution. As before, we see a steady deterioration in performance as the resource bound is decreased. The similar response times for resource bounds of 1000 and 500 suggest that the planner doesn't generally require as many as 1000 resource units to find a plan, and this is confirmed by the total absence of aborted runs even with a resource bound of 200.

Resource Bound	Av. Actions	Aborts	Mean Response Time
1000	25.8	0	599ms
500	26.5	0	627ms
200	28.1	0	521ms
100	40.9	1	347ms
50	46.6	7	199ms
10	51.6	12	44ms
2	55.1	10	11ms

Table 6: Results of Experiment `kw-3`

7 Concluding Remarks

How does the system described here differ from Bacchus and Kabanza's TLPlan, from which it takes its inspiration? First, TLPlan's temporal logic formulae have to be correct for all plans in order to preserve the completeness of the planner. Reactive rules, on the other hand, only supply recommendations. They can be completely wrong without threatening the planner's (stochastic) completeness. Second, TLPlan's heuristic formulae don't have a standalone role, and can't form the basis of a planner with anytime capability, like the one presented here.

The forward chaining approach to planning has many benefits that haven't been explored in this paper. For example, it's much easier to implement safety and maintenance goals in a forward-chaining mechanism than in a backward-chaining one. Similarly, it's easier to extend the planner to accept a rich input language, permitting concurrent actions, actions with indirect effects, and so on. These topics are the subject of ongoing work.

The forward-chaining approach to planning is further vindicated in the recent work of Hoffmann and Nebel [2001], whose forward-chaining FF planner outperforms other state-of-the-art domain-independent planners without relying on pre-defined domain-specific rules. Instead, the FF planner automatically extracts heuristics from the domain description to guide the forward search. A promising line of research would be to see

how such automatically generated heuristics could be used in combination with reactive rules in a system of the sort described here.

Acknowledgments

This work was funded by EPSRC project GR/N13104, "Cognitive Robotics II". Thanks to Paulo Santos.

References

[Agre & Chapman, 1987] P.Agre and D.Chapman, Pengi: An Implementation of a Theory of Activity, *Proceedings AAAI 87*, pp. 268–272.

[Bacchus & Kabanza, 2000] F.Bacchus and F.Kabanza, Using Temporal Logics to Express Search Control Knowledge for Planning, *Artificial Intelligence*, vol 116 (2000), pp. 123–191.

[Brooks, 1986] R.Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, Vol.2 (1986), pp. 14–23.

[Dean & Boddy, 1988] T.Dean and M.Boddy, An Analysis of Time-Dependent Planning, *Proceedings AAAI 88*, pp. 49–54.

[Drummond, 1989] M.Drummond, Situated Control Rules, *Proceedings KR 89*, pp. 103–113.

[Gat, 1992] E.Gatt, Integrating Planning and Reacting in a Heterogenous Asynchronous Architecture for Controlling Real-World Mobile Robots, *Proceedings AAAI 92*, pp. 809–815.

[Hoffmann & Nebel, 2001] J.Hoffmann and B.Nebel, The FF Planning System: Fast Plan Generation Through Heuristic Search, *Journal of Artificial Intelligence Research*, to appear.

[Kautz & Selman, 1999] H.Kautz and B.Selman, Unifying SAT-Based and Graph-Based Planning, *Proceedings IJCAI 99*, pp. 318–325.

[Korf, 1990] R.Korf, Real-Time Heuristic Search, *Artificial Intelligence*, Vol. 42 (1990), pp. 189–211.

[Mitchell, 1990] T.M.Mitchell, Becoming Increasingly Reactive, *Proceedings AAAI 90*, pp. 1051–1058.

[Pell, *et al.*, 1997] B.Pell, E.Gat, R.Keesing, N.Muscettola & B.Smith, Robust Periodic Planning and Execution for Autonomous Spacecraft, *Proceedings IJCAI 97*, pp. 1234–1239.

[Penberthy & Weld, 1992] J.S.Penberthy and D.S.Weld, UCPOP: A Sound, Complete, Partial Order Planner for ADL, *Proceedings KR 92*, pp. 103–114.

[Schoppers, 1987] M.J.Schoppers, Universal Plans for Reactive Robots in Unpredictable Environments, *Proceedings IJCAI 87*, pp. 1039-1046

[Zilberstein & Russell, 1993] S.Zilberstein and S.J.Russell, Anytime Sensing, Planning and Action: A Practical Model for Robot Control, *Proceedings IJCAI 93*, pp. 1402–1407.

On the Complexity of Planning in Transportation Domains

Malte Helmert

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee, Gebäude 052, 79110 Freiburg, Germany
helmert@informatik.uni-freiburg.de

Abstract. The efficiency of AI planning systems is usually evaluated empirically. The planning domains used in the competitions of the 1998 and 2000 AIPS conferences are of particular importance in this context. Many of these domains share a common theme of transporting *portables*, making use of *mobiles* traversing a map of *locations* and *roads*.
In this contribution, we embed these benchmarks into a well-structured hierarchy of *transportation problems* and study the computational complexity of optimal and non-optimal planning in this domain family. We identify the key features that make transportation tasks hard and try to shed some light on the recent success of planning systems based on heuristic local search, as observed in the AIPS 2000 competition.

1 Introduction

Apart from generally applicable hardness results [4], there is hardly any theoretical work on the time and space efficiency of common planning algorithms, so empirical methods have become the standard for performance evaluations in the planning community. Running time on problems from classical planning domains such as LOGISTICS and BLOCKSWORLD has often been (and still is) used for comparing the relative merits of planning systems. However, this kind of comparison is always difficult. If no planning system performs well in a given domain, does that mean that they are all poor, or is that domain intrinsically hard? If they all perform well, is this because of their strength or because of the simplicity of the task?

On a related issue, should planning systems be preferred that generate shorter plans but need more time? While there is no general answer to that question, theoretical results can contribute to the discussion, e. g. in cases where generating plans is easy but generating optimal plans is infeasible.

For addressing these issues, domain-specific complexity results for planning tasks appear to be useful. Pondering which domains to analyze, the ones that immediately spring to mind are the competition benchmarks from AIPS 1998 and AIPS 2000, considering their general importance for the planning community and the wealth of empirical performance data available.

While it would be possible to investigate each competition domain in isolation, it seems more worthwhile to identify commonly reoccurring concepts and

prove more general results that apply to domain families rather than individual domains. Not only does this help present the results in a more structured way, it also allows to shed some light on the *sources of hardness* in these benchmarks.

Because of space limitations, we only discuss the *transportation* domain family here, covering eight of the thirteen competition domains, namely GRID, GRIPPER, LOGISTICS, MYSTERY, MYSTERY', and three versions of MICONIC-10. A similar discussion of the other domains (ASSEMBLY, BLOCKSWORLD, FREECELL, MOVIE, and SCHEDULE) and the corresponding domain families as well as a more thorough discussion of the results presented here can be found elsewhere [8].

In the following section, we will introduce and analyze some new transportation problems generalizing most of the competition benchmarks. Section 3 applies the results of this analysis to the competition domains and covers some additional aspects of the GRID and MICONIC-10 domains. The implications of those results are discussed in Section 4, followed by some comments on related work in Section 5 and possible directions for future research in Section 6.

2 A Hierarchy of Transportation Problems

In this section, we will define and analyze a hierarchy of transportation problems that combines the key features of the important transportation benchmark domains.

Definition 1. TRANSPORT *task*
A TRANSPORT **task** *is a 9-tuple* $(V, E, M, P, fuel_0, l_0, l_G, cap, road)$*, where*

- (V, E) *is the* **roadmap** *graph; its vertices are called* **locations***, its edges are called* **roads***,*
- M *is a finite set of* **mobiles***,*
- P *is a finite set of* **portables** *(V, M, and P must be disjoint),*
- $fuel_0 : V \to \mathbb{N}$ *is the* **fuel** *function,*
- $l_0 : (M \cup P) \to V$ *is the* **initial location** *function,*
- $l_G : P \to V$ *is the* **goal location** *function,*
- $cap : M \to \mathbb{N}$ *is the* **capacity** *function, and finally*
- $road : M \to \mathbb{P}(E)$ *is the* **movement constraints** *function.*

This should not require much explanation. The goal location function is only defined for portables because we do not care about the final locations of mobiles. We do require that goal locations are specified for *all* portables, unlike most planning domains. This is because portables with unspecified goals could safely be ignored, not contributing to the hardness of the task.

The fuel function bounds the number of times a given location can be left by a mobile. Fuel is associated with locations rather than mobiles because this is the way it is handled in the MYSTERY-like domains. The carrying capacity function bounds the number of portables a given mobile can carry at the same time. The movement constraints function specifies which roads a given mobile is allowed to use.

We will now define some special cases of transportation tasks.

Definition 2. *Special cases of* TRANSPORT *tasks*

For $i, j \in \{1, \infty, *\}$ and $k \in \{1, +, *\}$, $\mathcal{I}_{ijk}$ is defined as the set of all TRANSPORT *tasks $I = (V, E, M, P, fuel_0, l_0, l_G, cap, road)$ satisfying:*

- *For $i = 1$, $cap(m) = 1$ for all mobiles m (one mobile can carry one portable).*
- *For $i = \infty$, $cap(m) = |P|$ for all mobiles m (unlimited capacity).*
- *For $j = 1$, $fuel_0(v) = 1$ for all locations v (one fuel unit per location).*
- *For $j = \infty$, $fuel_0(v) = \infty$[1] for all locations v (unlimited fuel).*
- *For $k = +$, $road(m) = E$ for all mobiles m (no movement restrictions).*
- *For $k = 1$, $road(m) = E$ for all mobiles m and $|M| = 1$ (no movement restrictions, only one mobile).*

According to this definition, the most general task set, containing all TRANSPORT tasks, is $\mathcal{I}_{***}$, and the most specific ones, having no proper specializations in the hierarchy, are $\mathcal{I}_{111}$, $\mathcal{I}_{1\infty1}$, $\mathcal{I}_{\infty11}$, and $\mathcal{I}_{\infty\infty1}$.

Definition 3. TRANSPORT *state transition graph*

The **state transition graph** $\mathcal{T}(I)$ *of a* TRANSPORT *task $I = (V, E, M, P, fuel_0, l_0, l_G, cap, road)$ is the digraph $(V_\mathcal{T}, A_\mathcal{T})$ with $V_\mathcal{T} = (M \cup P \to V \cup M) \times (V \to \{0, \ldots, \max fuel_0(V)\})$[2] and $((l, fuel), (l', fuel')) \in A_\mathcal{T}$ if and only if:*

$$
\begin{aligned}
(\exists m \in M, v, v' \in V: \quad & l(m) = v \wedge \{v, v'\} \in road(m) \wedge fuel(v) > 0 \\
& \wedge\ l' = l \oplus (m, v')^3 \wedge fuel' = fuel \oplus (v, fuel(v) - 1)) \\
\vee\ (\exists m \in M, p \in P: \quad & l(m) = l(p) \wedge |\{\ p \in P \mid l(p) = m\ \}| < cap(m) \\
& \wedge\ l' = l \oplus (p, m) \wedge fuel' = fuel) \\
\vee\ (\exists m \in M, p \in P: \quad & l(p) = m \wedge l' = l \oplus (p, l(m)) \wedge fuel' = fuel)
\end{aligned}
$$

This definition captures the intuition of legal state transitions in the specified transportation task. The first disjunct specifies transitions related to *movements* of a mobile, the second relates to a mobile *picking up* a portable, and the third to a mobile *dropping* a portable. In the following, we will only use these intuitive terms when talking about state transitions.

We can now define the decision problems we are interested in:

Definition 4. PLANEX-TRANSPORT$_{ijk}$

Given: TRANSPORT *task $I = (V, E, M, P, fuel_0, l_0, l_G, cap, road) \in \mathcal{I}_{ijk}$.*

Question: *In $\mathcal{T}(I)$, is there any directed path from $(l_0, fuel_0)$ to $(l_G, fuel')$ for some $fuel' \in V \to \mathbb{N}$?*

Definition 5. PLANLEN-TRANSPORT$_{ijk}$

Given: TRANSPORT *task $I = (V, E, M, P, fuel_0, l_0, l_G, cap, road) \in \mathcal{I}_{ijk}$, $K \in \mathbb{N}$.*

Question: *In $\mathcal{T}(I)$, is there a directed path of length at most K from $(l_0, fuel_0)$ to $(l_G, fuel')$ for some $fuel' \in V \to \mathbb{N}$?*

[1] Of course, ∞ is not a natural number. However, as we shall see shortly in the proof of Theorem 1, we can assume that there is "enough" fuel at each location, justifying this definition.

[2] States specify the location of mobiles and portables and the current fuel function.

[3] We use the notation $f \oplus (a', b')$ for *functional overloading*, i. e. the function f' with $f'(a') = b'$ and $f'(a) = f(a)$ for $a \neq a'$.

Theorem 1. PLANLEN-TRANSPORT$_{***} \in$ **NP**

Proof. If we can show that any solvable TRANSPORT task I has a solution of length $p(||I||)$ for some fixed polynomial p, then a simple guess and check algorithm can solve the problem non-deterministically.

This is true because each portable only needs to be at each location at most once, bounding the number of pickup and drop actions, and in between two pickup or drop actions, no mobile should visit a given location twice. □

Corollary 1. PLANEX-TRANSPORT$_{ijk} \leq_p$ PLANLEN-TRANSPORT$_{ijk}$ *for arbitrary values of* i, j, k *(and hence* PLANEX-TRANSPORT$_{***} \in$ **NP***)*

Proof. A TRANSPORT task I has a solution if and only if it has a solution of length $p(||I||)$, for the polynomial p from the preceding theorem. Therefore the mapping of I to $(I, p(||I||))$ is a polynomial reduction. □

2.1 Plan Existence

Theorem 2. PLANEX-TRANSPORT$_{*\infty *} \in$ **P**

Proof. Using breadth-first search on $road(m)$ starting at $l_0(m)$ for each mobile m with non-zero capacity, we can determine which roads can ever be used by any loaded mobile. The task can be solved if and only if for each portable p, $l_G(p)$ can be reached from $l_0(p)$ using these roads. This can easily be decided in polynomial time, and in fact the actual plans can easily be generated. □

This shows that the plan existence problems can be solved in polynomial time if no fuel constraints are present. We will now show that they are **NP**-complete otherwise, by proving **NP**-hardness of PLANEX-TRANSPORT$_{111}$ and PLANEX-TRANSPORT$_{\infty 11}$.

Theorem 3. PLANEX-TRANSPORT$_{111}$ *is* **NP***-complete*

Proof. Membership in **NP** is already known. We prove **NP**-hardness by a reduction from the **NP**-complete problem of finding a Hamiltonian path with a fixed start vertex [6, Problem GT39]. Let (V, E) be a graph and $v_1 \in V$. Then (V, E) contains a Hamiltonian path starting at v_1 if and only if there is a solution for the TRANSPORT task $I \in \mathcal{I}_{111}$ defined as follows: For each $v \in V$, there are two distinct locations v (called an *entrance*) and v^* (called an *exit*), with one unit of fuel each. At each entrance, there is a portable to be moved to the corresponding exit. There is only one mobile, of capacity one, starting at the entrance v_1. There are roads from v to v^* for $v \in V$ and from u^* to v for $\{u, v\} \in E$.

Now, if there is a Hamiltonian path in (V, E) starting at v_1, say $[v_1, \ldots, v_n]$, then there is a solution for the planning task where the movement path of the mobile is $[v_1, v_1^*, \ldots, v_n, v_n^*]$ and portables are picked up and dropped in the obvious way.

Now consider there is a solution to the planning task. Whenever a portable is picked up (at an entrance), the only reasonable thing to do is to move to its destination (the corresponding exit) and drop it, because there is no use in

deferring that movement when the carrying capacity is exhausted. The mobile must then proceed to the next entrance, which is only possible in the ways defined by the edges in the original graph. Thus, the plan corresponds to a path in the original graph that visits every vertex. It must be Hamiltonian, because if an entrance were ever visited twice, it could never be left again because of fuel constraints. □

Although the same reduction could be used in the infinite capacity case, we give another proof for this case showing that it is already **NP**-complete even if the roadmap is restricted to be a planar graph.

Theorem 4. PLANEX-TRANSPORT$_{\infty 11}$ *is* **NP***-complete*
PLANEX-TRANSPORT$_{\infty 11}$ *is* **NP***-complete, even if the roadmap is restricted to be a planar graph.*

Proof. Membership in **NP** is already known. For hardness, we reduce from the Hamiltonian Path problem with a fixed start vertex in a planar graph [8]. Let (V, E) be the graph and $v_1 \in V$. Then (V, E) contains a Hamiltonian path starting at v_1 if and only if there is a solution for the TRANSPORT task $I \in \mathcal{I}_{\infty 11}$ defined as follows: The roadmap of the planning task is (V, E), each location provides one unit of fuel, and there is one portable to be delivered to each location from v_1, the initial location of the only mobile (of unlimited capacity).

Clearly, this problem is solvable if and only if there is a Hamiltonian Path in (V, E) starting at v_1. □

This concludes our analysis of the PLANEX-TRANSPORT$_{ijk}$ decision problems. They can be solved in polynomial time if $j = \infty$ and are **NP**-complete otherwise.

2.2 Bounded Plan Existence

Theorems 1, 3 and 4 and Corollary 1 imply **NP**-completeness for PLANLEN-TRANSPORT$_{ijk}$ for $j \neq \infty$. In this subsection, we will show that the same result holds in the unrestricted fuel case, even in some very limited special cases.

In fact, the proofs of Theorems 3 and 4 can be adjusted to prove **NP**-completeness of PLANLEN-TRANSPORT$_{1\infty 1}$ and PLANLEN-TRANSPORT$_{\infty\infty 1}$ by replacing the fuel restrictions with plan length bounds of $4|V| - 1$ and $3|V| - 3$, respectively. However, these results require allowing for arbitrary (or arbitrary planar) roadmaps, and thus do not apply to planning domains such as LOGISTICS or GRID. For that reason, we will prove some stronger results now.

The first result in this section applies to *grid* roadmaps, i. e. graphs with vertex set $\{0, \ldots, w\} \times \{0, \ldots, h\}$ for some $w, h \in \mathbb{N}$ (called *width* and *height* of the grid, respectively), where vertices (a, b) and (a', b') are connected by an edge if and only if $|a - a'| + |b - b'| = 1$. Note that grids are always planar graphs.

Theorem 5. PLANLEN-TRANSPORT$_{1\infty 1}$ *is* **NP***-complete*
PLANLEN-TRANSPORT$_{1\infty 1}$ *is* **NP***-complete, even if the roadmap is restricted to be a grid.*

Proof. Membership in **NP** is already known. For hardness, we reduce from the $\mathcal{L}_1$ metric TSP, which is **NP**-complete in the strong sense [6, Problem ND23].[4]

Omitting the technical details which can be found elsewhere [8], the key idea is to have one portable for each site in the TSP instance, which needs to be moved to an adjacent location. The mobile starts at the northmost (with the highest y coordinate) TSP site and has to visit each site in order to deliver all portables, and the number of movements needed for that is equal to the length of the shortest non-closed TSP tour (i. e. a tour not returning to the initial location). The tour can be closed by putting an additional portable that needs to be moved "far up north".

To enforce that the length of the shortest plan is dominated by the movement between sites rather than movement between the initial and (adjacent) goal locations of portables the coordinates of the sites are scaled by a factor of $2n$ (n being the number of sites). □

The same reduction can be used in the unrestricted capacity case [8]. Additionally, in this setting the following result holds.

Theorem 6. PLANLEN-TRANSPORT$_{\infty\infty 1}$ *is* **NP**-*complete*

PLANLEN-TRANSPORT$_{\infty\infty 1}$ *is* **NP**-*complete, even if the roadmap is restricted to be a complete graph.*

Proof. Membership in **NP** is already known. For hardness, we reduce from the Feedback Vertex Set problem [6, Problem GT7]. Let (V, A) be a digraph and $K \in \mathbb{N}$. Then (V, A) contains a feedback vertex set of size at most K if and only if there is a solution of length at most $3|V| + 2|A| + K$ for the TRANSPORT task $I \in \mathcal{I}_{\infty\infty 1}$ where the roadmap is a complete graph with locations V and an additional location v_0, which is the initial location of the only mobile, there are no capacity or fuel constraints, there is one portable to be moved from v_0 to each $v \in V$ and one portable to be moved from u to v for each $(u, v) \in A$.

To see this, observe that for each feedback vertex set $V' \subseteq V$, the planning task can be solved by moving the mobile to the vertices from V' in any order, then to the vertices from $V \setminus V'$ in an order which is consistent with the arcs in the subgraph induced by $V \setminus V'$ (which must be acyclic because V' is a feedback vertex set), and finally to the vertices from V' again, in any order, picking up and dropping portables in the obvious way. This requires $|A| + |V|$ pickup and drop actions each and $|V| + |V'|$ movements, totaling a number of actions bounded by $3|V| + 2|A| + K$ if $|V'| \leq K$.

On the other hand, any plan must contain at least one pickup and drop action for each portable and visit each location at least once, totaling $3|V| + 2|A|$ actions, so if a plan does not exceed the given length bound, there cannot be more than K locations that are visited more than once. These locations must form a feedback vertex set. □

[4] Our transformation is only polynomial if numbers in the original TSP instance are encoded in unary, but this is a valid assumption for decision problems that are **NP**-complete in the strong sense.

3 Competition Domains from AIPS 1998/2000

Having completed the analysis of the TRANSPORT domain, we can now apply these results to the transportation domains from the planning competition.[5]

The MYSTERY domain [15] is equal to our $\mathcal{I}_{**+}$ task set. Thus, plan existence and bounded plan existence are **NP**-complete in this domain, even in the case of planar roadmaps, according to Theorems 1 and 4 and Corollary 1. This still holds if there is only one mobile and all portables start at the same location as the mobile.

The MYSTERY' domain [15] adds operators to move fuel between locations to the original MYSTERY domain. However, these can only be applied if at least two units of fuel are present at a given location, so for tasks from $\mathcal{I}_{*1+}$, there is no difference between the two domains and consequently the same hardness results apply for MYSTERY'. Membership in **NP** for the decision problems related to MYSTERY' follows from a polynomial plan length argument, as for the number of move, pickup and drop actions the same bounds as for TRANSPORT tasks apply, and there is no need to have more actions that move fuel than movements of mobiles.

LOGISTICS tasks [15] are special cases of $\mathcal{I}_{\infty\infty*}$ tasks and generalizations of $\mathcal{I}_{\infty\infty1}$ tasks with complete graph roadmaps. Thus, according to Theorems 1, 2, and 6, plans can be found in polynomial time in this domain, but the bounded plan existence problem is **NP**-complete, even if there is only one mobile (either truck or airplane).

The GRIPPER domain [15] is a specialization of $\mathcal{I}_{*\infty1}$ and thus allows for generating plans in polynomial time. Of course, this domain is so simple that even optimal plans can be generated in polynomial time.

For tasks without doors, the GRID domain [15] is very similar to $\mathcal{I}_{1\infty1}$ with grid roadmaps[6], thus the bounded plan existence problem in this domain is **NP**-hard, even in the absence of doors. It is actually in **NP** and thus **NP**-complete (with or without doors), again by a polynomial plan length argument, as it is not hard to bound the number of actions between two *unlock* actions in a reasonable plan, and no location can be unlocked more than once.

If optimality is not required, plans can be generated in the GRID domain in polynomial time by a simple strategy unlocking door after door as long as this is possible and then moving the keys to their goal destinations if reachable. [8]

Concluding our discussion of GRID, we want to briefly mention another proof of **NP**-hardness for the bounded plan existence problem without going into detail (cf. [8]). This reduction does not emphasize the route planning aspect of the domain and instead makes use of doors and is illustrated in Figure 1.

[5] Since a "benchmark domain" is not defined by the PDDL domain file alone (consider the LOGISTICS domain, where it is implicitly assumed that in well-formed problems the sets of portables, trucks and airplanes are disjoint), we refer to the literature for informal [1, 14, 15] and formal definitions [8] of these planning tasks.

[6] The only difference is that in GRID, the portable in hand can be swapped with a portable at the current location in just one action, but this does not make a difference for the proof of Theorem 5.

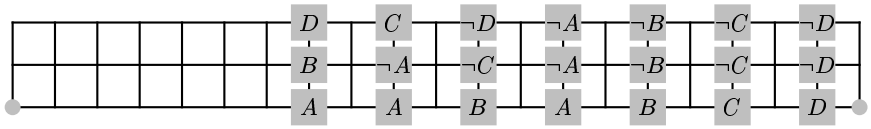


Fig. 1. GRID instance corresponding to $(A \vee B \vee D) \wedge (A \vee \neg A \vee C) \wedge (B \vee \neg C \vee \neg D)$. Locations with doors are marked with squares. The bottom left location contains the mobile and one key for each literal, opening the corresponding doors. The bottom right location contains an additional key. All keys must be moved to the bottom left location.

3.1 MICONIC-10

For the remaining competition domain, the MICONIC-10 elevator domain [12], things are a bit more complicated. There are actually three different domains under that name that were part of the AIPS 2000 competition. The first, called MICONIC-10 STRIPS, defines tasks very similar to LOGISTICS with one mobile, or $\mathcal{I}_{\infty\infty 1}$ with complete graphs. The only difference is that portables (passengers) can only be dropped at their destination locations and can never be picked up again (reboard the elevator). Theorems 1, 2, and 6 apply, and thus plans can be found in polynomial time, but deciding existence of a bounded length plan is an **NP**-complete problem.

The same is true for the second version of MICONIC-10, called *simple ADL* . In this version, all boarding and leaving at a given floor (picking up or dropping) is automatically handled by a single *stop* action with conditional effects. It causes all passengers inside the elevator with that goal destination to leave and all passengers waiting outside to board. This only requires a minor change to the proof of Theorem 6, changing the plan length bound to $2(|V|+K)+1$ for $|V|+K$ movements of the elevator and $|V|+K+1$ *stop* actions.

The "real" MICONIC-10 domain additionally introduces special passengers which impose movement restrictions on the elevator. Most importantly, the elevator may only stop at floors to which all passengers inside the elevator have access, and there are "attended" passengers who require the presence of at least one "attendant" passenger as long as they are inside the cabin (if the last attendant leaves the elevator, a new one must board). There are also VIP passengers who must be served with priority.

The decision problems related to that domain are still in **NP** because the number of stops can be bounded by twice the number of passengers to be served (one stop at their initial, another at their goal floor), and this in turn bounds the number of movements. However, as it turns out, plan existence is already **NP**-hard in this domain. Due to space restrictions, we will not give a formal definition of the decision problem at hand, which can be found in [8]. It should be possible to understand the following proof without those details, though.

Theorem 7. PLANEX-MICONIC-10 *is* **NP***-complete*

Proof. Membership in **NP** has been shown. For **NP**-hardness, we reduce from

the problem of finding a Hamiltonian path with a fixed start vertex v_1 in a digraph (V, A) [6, Problem GT39].

The corresponding MICONIC-10 task has the following floors: an *init floor* f_0, *final floor* f_∞, for each vertex u a *vertex start floor* f_u and *vertex end floor* f_u^*, and for each arc (u, v) an *arc floor* $f_{u,v}$. F is the set of all these floors and for each vertex u, F_u is the set containing f_u, f_u^*, and the arc floors for outgoing arcs of u. These are the passengers to be served:

Passenger	From	To	Access to ...	Special
p_0	f_0	f_{v_1}	$\{f_0, f_{v_1}\}$	VIP, attendant
$\forall u \in V$: p_u	f_0	f_u	$F \setminus \{f_\infty\}$	attended
$\forall u \in V$: p_u^*	f_u	f_u^*	$F_u \cup \{f_\infty\}$	attendant
$\forall u \in V$: p_u^∞	f_u^*	f_∞	$F \setminus \{f_u\}$	none
$\forall (u, v) \in A$: $p_{u,v}$	$f_{u,v}$	f_v	$\{f_{u,v}, f_u^*, f_v\}$	attendant

Assume that it is possible to solve the task. Because p_0 is a VIP, the first stops must be at f_0 and f_{v_1}, picking up all the attended passengers and $p_{v_1}^*$. Because of the movement restrictions of that passenger, the journey can only proceed to floors from F_{v_1}, and $f_{v_1}^*$ is not an option because going there would lead to the only attendant leaving. Thus, the elevator must go to f_{v_1,v_2} (for some vertex v_2 that is adjacent to v_1) and can then only proceed to $f_{v_1}^*$ and then f_{v_2}, picking up $p_{v_1}^\infty$.

We are now in a similar situation as upon arrival at f_{v_1}, and again, the elevator will eventually go to some floor f_{v_3}, then f_{v_4}, following the arcs of the digraph (V, A) in a path $[v_1, \ldots, v_n]$ until all vertices have been visited once. No vertex can be visited twice because of the passengers of type p_u^∞. So plan existence implies a Hamiltonian path starting at v_1 in the digraph.

On the other hand, if a Hamiltonian path exists, there is a sequence of elevator movements that leads to all attended passengers having arrived at their final destination and the elevator being at some floor f_u for $u \in V$. No longer requiring attendants, it can then immediately proceed to f_u^*, then f_∞ and finally serve the remaining passengers of type $f_{u,v}$ (for arcs (u, v) not part of the Hamiltonian path), one after the other, completing the plan. □

4 Discussion

Let us briefly summarize the results of our analysis. For fairly general transportation tasks, we have shown **NP**-completeness of non-optimal planning in the restricted fuel case and **NP**-completeness of optimal planning in all cases. Just finding some plan in tasks where fuel is abundant was shown to be a polynomial problem.

This is detailed in Figure 2. For some domains, even some severe restrictions are still sufficient to get **NP**-hardness. Specifically, all **NP**-hardness results in the multi-agent competition domains still hold if there is only one agent, and the **NP**-hardness result for GRID still holds if there are no doors at all. For convenience, we repeat the results for the competition domains:

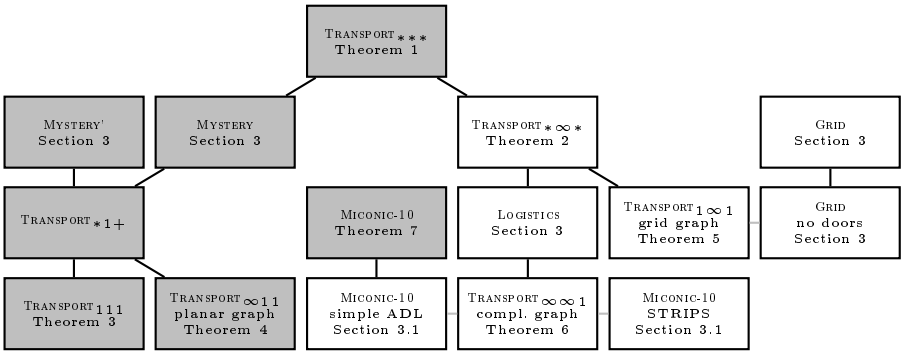


Fig. 2. The transportation domains hierarchy. Black lines indicate special cases, gray lines strong similarities of domains. Deciding plan existence is **NP**-complete for domains with gray boxes, plans can be generated in polynomial time for domains in white boxes. The bounded plan length problem is **NP**-complete for all domains in the figure. For the GRIPPER domain (not shown), both problems are polynomial.

Domain name	PLANEX	PLANLEN
GRID	polynomial	**NP**-complete
GRIPPER	polynomial	polynomial
LOGISTICS	polynomial	**NP**-complete
MICONIC-10 (STRIPS or simple ADL)	polynomial	**NP**-complete
MICONIC-10 (full ADL)	**NP**-complete	**NP**-complete
MYSTERY, MYSTERY'	**NP**-complete	**NP**-complete

It is interesting to observe that in those domains where heuristic local search planners such as **FF** [10] excel, the table lists different results for plan existence and bounded plan existence. Because all hardness proofs only use a single agent, they carry over to *optimal parallel planning*, which implies that in these domains planners like **Graphplan** [3] or **IPP** [11] try to solve provably hard subproblems that local search planners do not have to care about. When optimal plans are not required, local search has a conceptual advantage here, and we cannot hope for similar performance from any planner striving for optimality.

Greedy local search is less appropriate, however, if additional constraints can lead to dead ends in the search space. We have faced this problem when dealing with fuel constraints and in the full MICONIC-10 domain, where it may be unwise to have people board the elevator who restrict its movement too much. In fact, the competition domains with **NP**-hard plan existence problems are exactly the ones for which current planners based on heuristic local search encounter unrecognized dead ends.[7] [9]

While the observation that non-optimal planning is often easier than optimal planning is by no means surprising or new, we consider it important to point out. While there has been significant recent progress on non-optimal planning, optimal planners tend to get less attention than they deserve, maybe due to the

[7] This is also true for the non-transportation benchmarks. [8, 9]

fact that they are often compared to their non-optimal counterparts in terms of the size of problems they can handle. This kind of comparison is hardly fair.

We also observe that all discussed decision problems are in **NP**. We do not consider this a weakness of the benchmark set, as in STRIPS/ADL planning, **NP** membership is guaranteed as soon as there are polynomial bounds on plan lengths, which is a reasonable restriction from a plan execution point of view.

5 Related Work

Other work in the AI planning literature concerned with computational complexity results mostly focuses on domain-independent planning, analyzing different variants of the planning problem and special cases thereof [2, 4, 5]. This work mainly covers purely *syntactical* restrictions of general planning, such as limiting the number of operator preconditions or effects, but also discusses the complexity of STRIPS-style planning in (arbitrary) fixed domains [5].

There are very few articles in the planning literature which are concerned with the same kind of domain-dependent planning complexity results as this work. The existing literature concentrates on the complexity of BLOCKSWORLD, including results for generalizations of the classical domain, e. g. allowing for blocks of different size. The most comprehensive reference for this line of research is an article by Gupta and Nau [7]. There is also a very interesting discussion of the important distinction between optimal, near-optimal and non-optimal planning in BLOCKSWORLD. [16]

The usefulness of the idea of partitioning planning domains into families like *transportation* and most of the corresponding terminology is borrowed from work by Long and Fox [13], although in that paper the focus is on the automatic *detection* of transportation domains and the exploitation of some of their features by a planning algorithm, not on complexity aspects.

6 Outlook

While some questions were answered in the preceding sections, open issues remain. In some domains it would be interesting to investigate some more special cases to come up with more fine-grained results. For example, in the full MICONIC-10 domain, plan existence is **NP**-complete, but it is polynomial without special passengers and access restrictions. What is the complexity if only *some* of these enhancements are made?

Where plan existence is **NP**-complete, detecting the *phase transition* between (usually easy) under-constrained and (usually easy) over-constrained instances would be interesting, increasing the benefit of these domains for benchmarking.

Finally, in addition to discussing "optimal" and "non-optimal" planning, near-optimal planning is an interesting topic for domains where plan existence and bounded plan existence are of different complexity [17]. Giving performance guarantees is certainly easy in LOGISTICS and the restricted MICONIC-10 domains, but what about GRID?

References

1. Fahiem Bacchus and Dana S. Nau. The AIPS-2000 planning competition. *AI Magazine*, 2001. To appear.
2. Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.
3. Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.
4. Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.
5. Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):65–88, 1995.
6. Michael R. Garey and David S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness.* Freeman, 1979.
7. Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2–3):223–254, 1992.
8. Malte Helmert. On the complexity of planning in transportation and manipulation domains. Master's thesis, Albert-Ludwigs-Universität Freiburg, 2001. Postscript available at http://www.informatik.uni-freiburg.de/~ki/theses.html.
9. Jörg Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, 2001. Accepted for publication.
10. Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
11. Jana Köhler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, pages 273–285, New York, 1997. Springer-Verlag.
12. Jana Köhler and Kilian Schuster. Elevator control as a planning problem. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 331–338, 2000.
13. Derek Long and Maria Fox. Automatic synthesis and use of generic types in planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 196–205, 2000.
14. Derek Long, Henry Kautz, Bart Selman, Blai Bonet, Hector Geffner, Jana Köhler, Michael Brenner, Jörg Hoffmann, Frank Rittinger, Corin R. Anderson, Daniel S. Weld, David E. Smith, and Maria Fox. The AIPS-98 planning competition. *AI Magazine*, 21(2):13–33, 2000.
15. Drew McDermott. The 1998 AI Planning Systems competition. *AI Magazine*, 21(2):35–55, 2000.
16. Bart Selman. Near-optimal plans, tractability, and reactivity. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR'94)*, pages 521–529. Morgan Kaufmann, 1994.
17. John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125:119–153, 2001.

Short Papers

Generating Hard Satisfiable Scheduling Instances

Josep Argelich[1], Ramón Béjar[2], Alba Cabiscol[1], Cèsar Fernández[1], Felip Manyà[1], and Carla P. Gomes[2]

[1] Departament d'Informàtica i Enginyeria Industrial, Universitat de Lleida
Jaume II, 69, E-25001 Lleida, Spain
{alba,cesar,felip}@eup.udl.es
[2] Department of Computer Science, Cornell University
Ithaca, NY 14853, USA
{bejar,gomes}@cs.cornell.edu

Abstract. We present a random generator of partially complete round robin timetables that produces exclusively satisfiable instances, and provide experimental evidence that there is an easy-hard-easy pattern in the computational difficulty of completing partially complete timetables as the ratio of the number of removed entries to the total number of entries of the timetable is varied. Timetables in the hard region provide a suitable test-bed for evaluating and fine-tuning local search algorithms.

1 Introduction

Local search algorithms (LSA's) are widely used to efficiently solve planning and scheduling problems [3, 9]. One difficulty with LSA's is that they are incomplete and cannot prove unsatisfiability. Thus, benchmark instances for measuring the performance of LSA's have to be satisfiable. Unfortunately, it has proven to be surprisingly difficult to develop random generators of hard satisfiable instances of combinatorial problems [1].

Given a set of candidate benchmark instances, unsatisfiable instances are generally filtered out with complete algorithms, and then only satisfiable instances are used to evaluate and fine-tune LSA's. However, this approach is problematic in problems where incomplete algorithms can solve larger instances than complete algorithms because the latter cannot identify hard satisfiable instances.

In this paper we describe a random generator of satisfiable scheduling instances which are computationally difficult to solve with LSA's for SAT. Our generator starts by randomly creating a timetable T for a temporally dense single round robin tournament using the incomplete satisfiability solver *Walk-SAT* [11]. Then, it generates a partially complete round robin timetable T' by randomly removing a given number of entries of T in such a way that the number of removed entries in each column and each row are approximately equal. The underlying generation model guarantees that T' can be completed into a feasible timetable, and has the advantage that the expected hardness of completing a

partially complete timetable can be finely controlled by tuning the number of removed entries.

In order to investigate the hardness of the instances of our generator we conducted a comprehensive experimental investigation. We observed that there is an easy-hard-easy pattern in the computational difficulty of completing partially complete timetables with LSA's for SAT as the ratio of the number of removed entries to the total number of entries is varied. Timetables in the hard region provide a suitable set of randomly generated satisfiable scheduling benchmarks.

We considered the generic problem solving approach that consists in modeling combinatorial problems as SAT instances and then solving the resulting instance with a SAT solver. In the last years the planning as satisfiability approach has gained popularity and has allowed the creation of planning systems like *Blackbox* [9]. The scheduling as satisfiability approach was used by Crawford & Baker [5] to solve the job shop problem and by Béjar & Manyà [2, 3] to create timetables for a variant of round robin tournaments. The generation of satisfiable instances for the quasigroup completion problem was investigated by Achlioptas, Gomes, Kautz & Selman [1, 6], as well as in [8]. Their papers inspired our work on the round robin completion problem.

The paper is structured as follows. In Section 2 we introduce the round robin problem. In Section 3 we describe the random generator of partially complete timetables. In Section 4 we present and discuss the experimental investigation.

2 The round robin problem

In this paper we consider the timetabling problem for temporally dense single round robin tournaments (DSRR): given an even number of teams n, the DSRR problem consists in distributing $n(n-1)/2$ matches in $n-1$ rounds in such a way that each team plays each other team exactly once during the competition. Figure 1 shows a 6-team DSRR timetable. We represent DSRR timetables for n teams by an $n \times (n-1)$ matrix o of variables, where variables $o_{t,r}$ tell the opponent team against which team t plays in round r.

The DSRR problem for n teams can be represented as a constraint satisfaction problem (CSP) [7] as follows:

- The set of variables is formed by all variables $o_{t,r}$ in matrix o.
- The domain $D_{o_{t,r}}$ of each variable $o_{t,r}$ is $\{1, \ldots, n\}$.
- The set of constraints is formed by the following constraints:
  - `all-different`$(o_{t,1}, \ldots, o_{t,n-1})$, for every $t \in \{1, \ldots, n\}$, and
  - `round-matches`$(o_{1,r}, \ldots, o_{n,r})$, for every $r \in \{1, \ldots, n-1\}$.

The constraints `all-different` and `round-matches` are defined as follows:

$$\texttt{all-different}(x_1, \ldots, x_m) = \{(v_1, \ldots, v_m) \in D_{x_1} \times \cdots \times D_{x_m} | \forall_{i,j,i \neq j}\, v_i \neq v_j\}$$

$$\texttt{round-matches}(x_1, \ldots, x_m) = \{(v_1, \ldots, v_m) \in D_{x_1} \times \cdots \times D_{x_m} | \forall_{i,j,i \neq j}\, v_i \neq i \wedge v_i = j \leftrightarrow v_j = i\}$$

teams/rounds	1	2	3	4	5
1	2	4	6	3	5
2	1	3	5	6	4
3	5	2	4	1	6
4	6	1	3	5	2
5	3	6	2	4	1
6	4	5	1	2	3

Fig. 1. A 6-team DSRR timetable

	1	2	3	4	5	6
1	X	1	4	2	5	3
2	1	X	2	5	3	4
3	4	2	X	3	1	5
4	2	5	3	X	4	1
5	5	3	1	4	X	2
6	3	4	5	1	2	X

Fig. 2. A symmetric quasigroup

The `all-different` constraint expresses that each row of a DSRR timetable contains every team only once, and the `round-matches` constraint expresses that each column groups the teams into matches; each column represents all the matches of one round.

Next, we define the SAT encoding of the n-team DSRR problem that we used in the experimental investigation described in Section 4.

1. The set $\{p^k_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq n-1, 1 \leq k \leq n, i \neq k\}$ is the set of propositional variables. The intended meaning of p^k_{ij} is that team i plays against team k in round j.
2. The constraint $\texttt{all-different}(p_{i,1}, \ldots, p_{i,n-1})$ is defined as follows:

$$\bigwedge_{1 \leq j < n} \Big(\bigvee_{\substack{1 \leq k \leq n \\ k \neq i}} p^k_{i,j} \quad \wedge \bigwedge_{\substack{1 \leq k_1 < k_2 \leq n \\ k_1 \neq k_2 \neq i}} (\neg p^{k_1}_{i,j} \vee \neg p^{k_2}_{i,j}) \Big) \quad \wedge$$

$$\bigwedge_{\substack{1 \leq j_1 < j_2 < n \\ j_1 \neq j_2}} \bigwedge_{\substack{1 \leq k \leq n \\ k \neq i}} (\neg p^k_{i,j_1} \vee \neg p^k_{i,j_2})$$

3. The constraint $\texttt{round-matches}(p_{1,j}, \ldots, p_{n,j})$ is defined as follows:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{\substack{1 \leq k \leq n \\ k \neq i}} (\neg p^k_{i,j} \vee p^i_{k,j})$$

We define the *DSRR completion problem* to be the problem of determining whether a partially complete DSRR timetable can be completed into a feasible timetable. In Section 4 we provide experimental evidence that completing a DSRR timetable is computationally harder than constructing a full timetable.

The DSRR completion problem is NP-complete. This follows from the fact that it is equivalent to the problem of completing partially complete symmetric quasigroups, which is know to be NP-complete [4]. Figure 2 shows the symmetric quasigroup of size 6 that corresponds to the DSRR timetable of Figure 1. We use the symbols $\{1, 2, 3, 4, 5, X\}$ to fill the entries of the quasigroup: the entry in row i and column j is $r \in \{1, 2, 3, 4, 5\}$ if team i plays against team j in round r, and the entries of the diagonal of the quasigroup are X.

3 A random generator of partially complete timetables

The random generator of partially complete DSRR timetables that we have designed and implemented has two peculiarities: (i) produces exclusively satisfiable instances, and (ii) the number of removed entries in each column and each row are approximately equal. It has been shown recently that removing entries in a balanced way allows one to generate hard quasigroup completion problems [8].

The pseudo-code is shown in Figure 3: we represent entries by $o^{t'}_{t,r}$ and refer to removed entries as *holes*. The intended meaning of $o^{t'}_{t,r}$ is that team t plays against team t' in round r.

procedure Random-Generator
input: an even number of teams n and an even number of holes h
output: an n-team partially complete timetable with h holes

$h' := \lfloor \frac{h}{n-1} \rfloor + 1$
$T :=$ a randomly generated n-team complete DSRR timetable
`while` $h > 0$ `do`
  $S :=$ set of non-empty entries $o^{t'}_{t,r}$ of T such that rows t, t' have
    have less than $h' + 1$ holes and column r has less than h' holes;
  $o^{t'}_{t,r} :=$ a randomly selected entry of S;
  $T := T$ with entries $o^{t'}_{t,r}$ and $o^{t}_{t',r}$ removed;
  $h := h - 2$;
`endwhile`
`return`(T);

Fig. 3. Random generator of partially complete timetables

In the experimental investigation, we used *WalkSAT* and the SAT encoding defined in Section 2 to randomly generate a complete DSRR timetable. SAT encodings of partially complete timetables were obtained by adding the list of holes to the SAT encoding of the corresponding complete timetable. As the resulting encodings had a considerable number of unit clauses, they were first simplified by applying unit propagation and then solved with *WalkSAT*.

It is worth mentioning that *WalkSAT* takes less than 1 minute to find a complete timetable for 40 teams. Systematic satisfiability algorithms like *Satz* [10] are not able to solve complete DSRR timetable for 14 teams after 48 hours.

4 Experimental results

In the experimental investigation we first used the random generator of partially complete timetables to produce sets of instances for different number of teams: $n = 30, 32, 36$; we considered these values of n in order to get experimental results in a reasonable amount of time. For all the sets, we varied the ratio of the number of holes (h) to the total number of entries of the timetable ($n \times (n-1)$) from 0.350 to 0.420 for $n = 30, 32$, and from 0.320 to 0.400 for $n = 36$; we incremented that ratio by 0.001 in each step. At each setting we ran *WalkSAT* on 20 partially

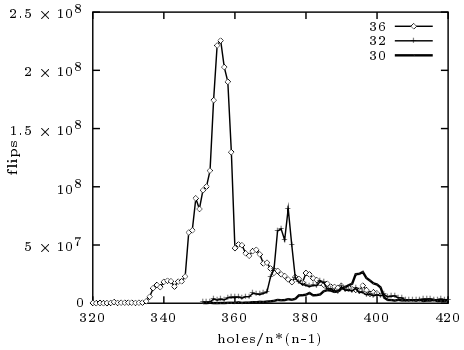


Fig. 4. Computational cost profiles for 30, 32 and 36 teams

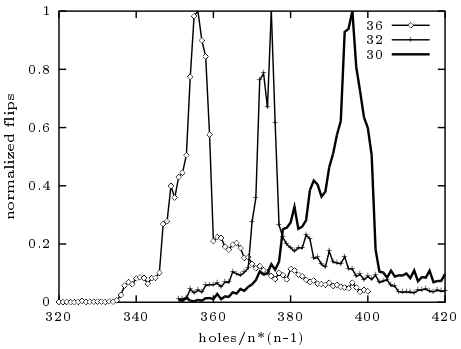


Fig. 5. Normalized computational cost profiles for 30, 32 and 36 teams

complete timetables. Each instance was executed until 25 solutions were found using no cutoff (maxflips), 30% noise for $n = 30$, 26% noise for $n = 32$, and 20% noise for $n = 36$. We used approximately optimal noise parameter settings for each timetable size. Such experiments were performed on PC's with 500 MHz Pentium III Processors under Linux Operating System.

Figure 4 visualizes the easy-hard-easy pattern in the computational difficulty of completing partially complete timetables with *WalkSAT* for $n = 30, 32, 36$. Along the horizontal axis is the ratio of the number of holes to the total number of entries of the timetable, and along the vertical axis is the median number of flips needed to solve an instance.

Figure 5 is like Figure 4 but the median number of flips are normalized. One can observe that there is a shift in the location of the hardness peak as a function of the number of teams.

Figure 6 visualizes the easy-hard-easy pattern for $n = 32$ showing seconds instead of flips. Along the vertical axis are the mean and median number of seconds needed to solve an instance.

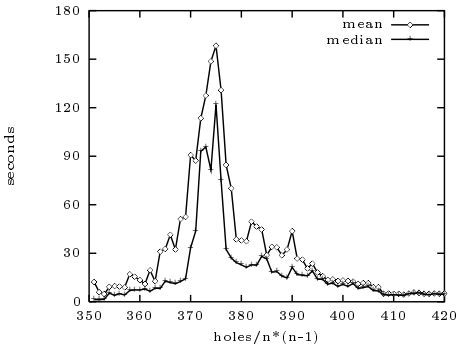


Fig. 6. Mean and median number of seconds for 32 teams

From the experimental results we can conclude that, when we use our random generator of partially complete timetables, there is an easy-hard-easy pattern in the computational difficulty of completing partially complete timetables with LSA's for SAT as the ratio of the number of removed entries to the total number of entries is varied. Thus, the expected hardness of completing a timetable can be finely controlled by tuning the number of removed entries, and timetables in the hard region provide a source of suitable scheduling benchmarks to evaluate and fine-tune LSA's.

Taking into account the existing work on the quasigroup completion problem [1, 8], and the equivalence between the DSRR completion problem and the problem of completing partially complete symmetric quasigroups, we conjecture that the easy-hard-easy pattern could also be observed if we use non-Boolean encodings as well as algorithms other than *WalkSAT*. A crucial point for obtaining this difficulty profile was the generation model of partially complete timetables. We did not identify the easy-hard-easy pattern with other strategies of *making holes*.

Acknowledgements: Research partially supported by the DARPA contracts F30602-00-2-0530 and F30602-00-2-0596, and project CICYT TIC96-1038-C04-03. The fifth author was supported by grant PR2001-0163 of the "Secretaría de Estado de Educación y Universidades".

References

1. D. Achlioptas, C. P. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proc. of AAAI-2000*, pages 256–261, 2000.
2. R. Béjar and F. Manyà. Solving combinatorial problems with regular local search algorithms. In *Proc. of LPAR'99*, pages 33–43. Springer LNAI 1705, 1999.
3. R. Béjar and F. Manyà. Solving the round robin problem using propositional logic. In *Proc. of AAAI-2000*, pages 262–266, 2000.
4. C. Colbourn. Embedding partial steiner triple systems is NP-complete. *Journal of Combinatorial Theory, Series A*, 35:100–105, 1983.
5. J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of AAAI'94*, pages 1092–1097, 1994.
6. C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proc. of AAAI'97*, pages 221–226, 1997.
7. M. Henz, T. Müller, S. Thiel, and M. van Brandenburg. Global constraints for round robin tournament scheduling, 2001. Draft paper.
8. H. A. Kautz, Y. Ruan, D. Achlioptas, C. P. Gomes, B. Selman, and M. Stickel. Balance and filtering in structured satisfiable problems. In *Proc. of IJCAI'01*, 2001.
9. H. A. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proc. of IJCAI'99*, pages 318–325, 1999.
10. C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proc. of CP'97*, pages 341–355. Springer LNCS 1330, 1997.
11. B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. of AAAI'94*, pages 337–343, 1994.

Learning Robot Action Plans for Controlling Continuous, Percept-driven Behavior *

Michael Beetz[1] and Thorsten Belker[2]

[1] Computer Science Dept. IX, Munich University of Technology, Munich, Germany
[2] Dept. of Computer Science III, University of Bonn, Bonn, Germany

Abstract. Autonomous robots, such as robot office couriers, need control routines that support flexible task execution and effective action planning. This paper describes XFRMLEARN, a system that learns structured symbolic robot action plans for navigation tasks. Given a navigation task, XFRMLEARN learns to structure continuous navigation behavior and represents the learned structure as compact and transparent plans. The structured plans are obtained by starting with monolithical default plans that are optimized for average performance and adding subplans to improve the navigation performance for the given task. Compactness is achieved by incorporating only subplans that achieve significant performance gains. The resulting plans support action planning and opportunistic task execution. XFRMLEARN is implemented and extensively evaluated on an autonomous mobile robot.

1 Introduction

Robots operating in human working environments and solving dynamically changing sets of complex tasks are challenging testbeds for autonomous robot control. The dynamic nature of the environments and the nondeterministic effects of actions requires robots to exhibit concurrent, percept-driven behavior to reliably cope with unforeseen events. Most physical control processes are continuous in nature and difficult to represent in discrete symbolic structures.

In order to apply high-level plan-based control techniques, robots need adequate and realistic representations of their control processes that enable their planning routines to foresee problems and forestall them. In this paper we take the navigation behavior of autonomous mobile robots as our prototypical example.

Different approaches have been proposed to specify the navigation behavior of such robots. A number of researchers consider navigation as an instance of Markov decision problems (MDPs) [KCK96]. They model the navigation behavior as a finite state automaton in which navigation actions cause stochastic state transitions. The robot is rewarded for reaching its destination quickly and reliably. A solution for such problems is a *policy*, a mapping from discretized robot poses into fine-grained navigation actions.

MDPs form an attractive framework for navigation because they use a uniform mechanism for action selection and a parsimonious problem encoding. The navigation policies computed by MDPs aim at robustness and optimizing the average performance. One

* This research is partially funded by the Deutsche Forschungsgemeinschaft.

of the main problems in the application of MDP planning techniques is to keep the state space small enough so that the MDPs are still solvable.

Another approach is the specification of environment- and task-specific navigation plans, such as ***structured reactive navigation plans (*SRNPs*)*** [Bee99]. SRNPs specify a default navigation behavior and employ additional concurrent, percept-driven subplans that overwrite the default behavior while they are active. The default navigation behavior can be generated by an MDP navigation system. The (de-)activation conditions of the subplans structure the continuous navigation behavior in a task-specific way.

SRNPs are valuable resources for opportunistic task execution and effective action planning because they provide high-level controllers with subplans such as traverse a particular narrow passage or an open area. More specifically, SRNPs (1) can generate qualitative events from continuous behavior, such as entering a narrow passage; (2) support online adaptation of the navigation behavior (drive more carefully while traversing a particular narrow passage), and (3) allow for compact and realistic symbolic predictions of continuous, sensor-driven behavior. The specification of good task and environment-specific SRNPs, however, requires tailoring their structure and parameterizations to the specifics of the environment.

We propose to bridge the gap between both approaches by learning SRNPs, symbolic plans, from executing MDP navigation policies. Our thesis is that a robot can autonomously learn compact and well-structured SRNPs by using MDP navigation policies as default plans and repeatedly inserting subplans into the SRNPs that significantly improve the navigation performance. This idea works because the policies computed by the MDP path planner are already fairly general and optimized for average performance. If the behavior produced by the default plans were uniformly good, making navigation plans more sophisticated would be of no use. The rationale behind requiring subplans to achieve significant improvements is to keep the structure of the plan simple.

2 An Overview on XFRMLEARN

We have implemented XFRMLEARN a realization of this learning model and applied it to learning SRNPs for an autonomous mobile robot that is to perform office courier service. XFRMLEARN is embedded into a high-level robot control system called ***structured reactive controllers*** (SRCs) [Bee99]. SRCs are are controllers that can revise their intended course of action based on foresight and planning at execution time. SRCs employ and reason about plans that specify and synchronize ***concurrent percept-driven*** behavior. Concurrent plans are represented in a transparent and modular form so that automatic planning techniques can make inferences about them and revise them.

XFRMLEARN is applied to the RHINO navigation system [BCF+00], which has shown impressive results in several longterm experiments. Conceptually, this robot navigation system works as follows. A navigation problem is transformed into a Markov decision problem to be solved by a path planner using a value iteration algorithm. The solution is a policy that maps every possible location into the optimal heading to reach the target. This policy is then given to a reactive collision avoidance module that executes the policy taking the actual sensor readings into account [BCF+00].

The RHINO navigation system can be parameterized in different ways. The parameter PATH is a sequence of intermediate points which are to be visited in the specified order. COLLI-MODE determines how cautiously the robot should drive and how abruptly it is allowed to change direction. RHINO's navigation behavior can be improved because RHINO's path planner solves an idealized problem that does not take the desired velocity, the dynamics of the robot, the sensor crosstalk, and the expected clutteredness fully into account. The reactive collision avoidance component takes these aspects into account but makes only local decisions.

We propose an "analyze, revise, and test" cycle as a computational model for learning SRNPs. XFRMLEARN starts with a default plan that transforms a navigation problem into an MDP problem and passes the MDP problem to RHINO's navigation system. After RHINO's path planner has determined the navigation policy the navigation system activates the collision avoidance module for the execution of the resulting policy. XFRMLEARN records the resulting navigation behavior and looks for stretches of behavior that could be possibly improved. XFRMLEARN then tries to explain the improvable behavior stretches using causal knowledge and its knowledge about the environment. These explanations are then used to index promising plan revision methods that introduce and modify subplans. The revisions are then tested in a series of experiments to decide whether they are likely to improve the navigation behavior. Successful subplans are incorporated into the symbolic plan.

3 Structured Reactive Navigation Plans

Let us now take a more detailed look at the representation of SRNPs.

navigation plan *(desk-1,desk-2)*
  with subplans
    TRAVERSE-NARROW-PASSAGE(⟨*635,1274*⟩,⟨*635,1076*⟩)
      parameterizations *colli-mode* ← *slow*
      path constraints ⟨*635,1274*⟩,⟨*635,1076*⟩
      justification *narrow-passage-bug-3*
    TRAVERSE-NARROW-PASSAGE(...)
    TRAVERSE-FREE-SPACE(...)
  DEFAULT-GO-TO (*desk-2*)

The SRNP above contains three subplans: one for leaving the left office, one for entering the right one, and one for speeding up the traversal of the hallway. The subplan for leaving the left office is shown in more detail. The path constraints are added to the plan for causing the robot to traverse the narrow passage orthogonally with maximal clearance. The parameterizations of the navigation system specify that the robot is asked to drive slowly in the narrow passage and to only use laser sensors for obstacle avoidance to avoid the hallucination of obstacles due to sonar crosstalk.

SRNPs are called ***structured*** because the subplans explicitly represent task-relevant structure in continuous navigation behavior. They are called ***reactive*** because "perceived" qualitative events, such as entering or leaving a narrow passage, trigger the activation and termination of subplans.

4 XFRMLEARN in Detail

A key problem in learning structured navigation plans is to structure the navigation behavior well. Because the robot must start the subplans and synchronize them, it must be capable of "perceiving" the situations in which subplans are to be activated and deactivated. Besides being perceiveable, the situations should be relevant for adapting navigation behavior. Among others, we use the concept of ***narrowness*** for detecting the situations in which the navigation behavior is to be adapted. Based on the concept of narrowness the robot can differentiate situations such as free space, traversing narrow passages, entering narrow passages, and leaving narrow passages.

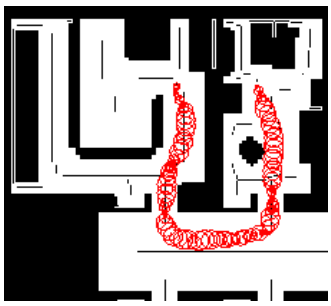


Fig. 1: Visualization of a behavior trace. The center of the circles denote the robot's position and the size of the circle the current speed of the robot.

Diagnosis of Conspicous Subtraces. Upon carrying out a navigation task, RHINO produces a behavior trace like the one shown in Figure 1. The robot's position is depicted by a circle where the size of the circle is proportional to the robot's translational speed. Behavior feature subtraces such as low and high translational speed, turning in place, and frequent stopping often hint at which behavior stretches can be improved. To infer how the improvements might be achieved, XFRMLEARN first tries to explain a behavior feature subtrace by finding environment feature subtraces, such as "traversing a narrow passage" or "passing an obstacle" that overlap with it. We use the predicate MAYCAUSE(F1,F2) to specify that the environment feature F1 may cause the behavior feature F2 like narrow passages causing low translational velocity. If there is sufficient overlap between a behavior feature subtrace *b* and an environment feature subtrace *e* then the behavior is considered to be a ***behavior flaw***. The diagnosis step is realized through a simple diagnostic rule that, depending on the instantiation of MAYCAUSE(?F1,?F2) can diagnose different kinds of flaws:

D-1 Low translational velocity is caused by the traversal of narrow passages.
D-2 Stopping is caused by the traversal of narrow passage.
D-3 Low translational velocity is caused by passing an obstacle too close.
D-4 Stopping caused by passing an obstacle too close.
D-5 High target velocity caused by traversing free space.

The "revise" step uses programming knowledge about how to revise navigation plans and how to parameterize the subsymbolic navigation modules that is encoded in the form of plan transformation rules. In their condition parts transformation rules check their applicability and the promise of success. These factors are used to estimate the expected utility of rule applications. XFRMLEARN selects the rule with a probability proportional to the expected utility and applies it to the plan.

For the purpose of this paper, XFRMLEARN provides the following revisions:

R-1 If the behavior flaw is attributed to the traversal of a narrow passage then insert a subplan to traverse the passage orthogonally and with maximal clearance.
R-2 Switch off the sonar sensors while traversing narrow passages if the robot repeatedly stops during the traversal.

R-3 Insert an additional path constraint to pass a closeby obstacle with more clearance.
R-4 Increase the target velocity for the traversal of free space where the measured velocity almost reaches the current target velocity.
R-5 Insert an additional path constraint to avoid abrupt changes in the robot's heading.

Because XFRMLEARN's transformation rules are heuristic, their applicability and the performance gain that can be expected from their application is environment and task-specific. Therefore XFRMLEARN learns the environment and task specific expected utility of rules based on experience.

The "test" step. Because plan transformation rules check their applicability and parameterization with respect to idealized models of the environment, the robot, the perceptual apparatus, and operation of the subsymbolic navigation system, XFRMLEARN cannot guarantee any improvements of the existing plan. Therefore, XFRMLEARN tests the resulting candidate plans against the original plan by repeatedly running the original and the revised plan and measuring the time performance in the local region that is affected by the plan transformation. The new candidate plan is accepted, if based on the experiments there is a 95% confidence that the new plan performs better than the original one.

5 Experimental Results

To empirically evaluate XFRMLEARN we have performed two long term experiments in which XFRMLEARN has improved the performance of the RHINO navigation system for given navigation tasks by up to 44 percent within 6 to 7 hours.

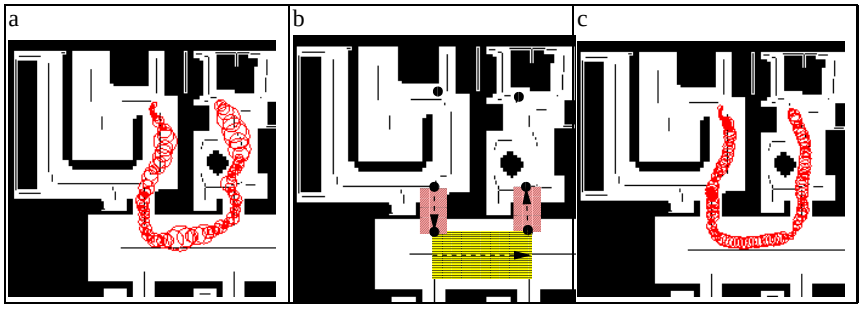


Fig. 2. Behavior trace of the default plan (a). Low T-Vel subtraces (b). Learned SRNP (c).

Figure 2(a) shows the navigation task (going from the desk in the left room to the one in the right office) and a typical behavior trace generated by the MDP navigation system. Figure 2(b) visualizes the plan that was learned by XFRMLEARN. It contains three subplans. One for traversing the left doorway, one for the right one, and one for the traversal of the hallway. The ones for traversing the doorways are TRAVERSENARROWPASSAGE subplans, which comprise path constraints (the black circles) as well as behavior adaptations (depicted by the region). The subplan is activated when the region is entered and deactivated when it is left. A typical behavior trace of the learned

SRNP is shown in Figure 2(c). We can see that the behavior is much more homogeneous and that the robot travels faster. This visual impression is confirmed by statistical tests. The t-test for the learned SRNP being at least 24 seconds (21%) faster returns a significance of 0.956. A bootstrap test returns the probability of 0.956 that the variance of the performance has been reduced.

In the second learning session the average time needed for performing a navigation task has been reduced by about 95.57 seconds (44%). The t-test for the revised plan being at least 39 seconds (18%) faster returns a significance of 0.952. A bootstrap test returns the probabilty of 0.857 that the variance of the performance has been reduced.

6 Conclusions

We have described XFRMLEARN, a system that learns SRNPs, symbolic behavior specifications that (a) improve the navigation behavior of an autonomous mobile robot generated by executing MDP navigation policies, (b) make the navigation behavior more predictable, and (c) are structured and transparent so that high-level controllers can exploit them for demanding applications such as office delivery.

XFRMLEARN is capable of learning compact and modular SRNPs that mirror the relevant temporal and spatial structures in the continuous navigation behavior because it starts with default plans that produce flexible behavior optimized for average performance, identifies subtasks, stretches of behavior that look as if they could be improved, and adds subtask specific subplans only if the subplans can improve the navigation behavior significantly.

The learning method builds a synthesis among various subfields of AI: computing optimal actions in stochastic domains, symbolic action planning, learning and skill acquisition, and the integration of symbolic and subsymbolic approaches to autonomous robot control. Our approach also takes a particular view on the integration of symbolic and subsymbolic control processes, in particular MDPs. In our view symbolic representations are resources that allow for more economical reasoning. The representational power of symbolic approaches can enable robot controllers to better deal with complex and changing environmants and achieve changing sets of interacting jobs. This is achieved by making more information explicit and representing behavior specifications symbolically, transparently, and modularly. In our approach, (PO)MDPs are viewed as a way to ground symbolic representations.

References

[BCF+00] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *ArtificialIntelligence*, 114(1-2), 2000.

[Bee99] M. Beetz. Structured reactive controllers — a computational model of everyday activity. In O. Etzioni, J. Müller, and J. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents*, pages 228–235, 1999.

[KCK96] L. Kaelbling, A. Cassandra, and J. Kurien. Acting under uncertainty: Discrete bayesian models for mobile-robot navigation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1996.

Reinforcement Learning for Weakly-Coupled MDPs and an Application to Planetary Rover Control

Daniel S. Bernstein and Shlomo Zilberstein

Department of Computer Science, University of Massachusetts,
Amherst, Massachusetts 01003
{bern, shlomo}@cs.umass.edu

Abstract. Weakly-coupled Markov decision processes can be decomposed into subprocesses that interact only through a small set of bottleneck states. We study a hierarchical reinforcement learning algorithm designed to take advantage of this particular type of decomposability. To test our algorithm, we use a decision-making problem faced by autonomous planetary rovers. In this problem, a Mars rover must decide which activities to perform and when to traverse between science sites in order to make the best use of its limited resources. In our experiments, the hierarchical algorithm performs better than Q-learning in the early stages of learning, but unlike Q-learning it converges to a suboptimal policy. This suggests that it may be advantageous to use the hierarchical algorithm when training time is limited.

1 Introduction

The Markov decision processes (MDP) framework is widely used to model problems in decision-theoretic planning and reinforcement learning [6]. Recently there has been increased interest in delimiting classes of MDPs that are naturally decomposable and developing special-purpose techniques for these classes [1]. In this paper, we focus on reinforcement learning for weakly-coupled MDPs. A weakly-coupled MDP is an MDP that has a natural decomposition into a set of subprocesses. The transition from one subprocess to another requires entry into one of a small set of bottleneck states. Because the subprocesses are only connected through a small set of states, they are "almost" independent. The common intuition is that weakly-coupled MDPs should require less computational effort to solve than arbitrary MDPs.

The algorithm that we investigate is a reinforcement learning version of a previously studied planning algorithm for weakly-coupled MDPs [2]. The planning algorithm is model based, whereas our algorithm requires only information from experience trajectories and knowledge about which states are the bottleneck states. This can be beneficial for problems where only a simulator or actual experience are available. Our algorithm fits into the category of hierarchical reinforcement learning (see, e.g., [7]) because it learns simultaneously at the state level and at the subprocess level. We note that other researchers have proposed methods for solving weakly-coupled MDPs [3–5], but very little work has been done in a reinforcement learning context.

For experimentation we use a problem from autonomous planetary rover control that can be modeled as a weakly-coupled MDP. In our decision-making scenario, a rover on

Mars must explore a number of sites over the course of a day without stopping to establish communication with Earth. Using only a list of sites, information about its resource levels, and information about the goals of the mission, the rover must decide which activities to perform and when to move from one site to the next. Limited resources and nondeterministic action effects make the problem nontrivial. In the main body of the paper, we describe in detail how this problem can be modeled as a weakly-coupled MDP, with each site being a separate subprocess.

We compare the hierarchical algorithm with Q-learning, and we see that the hierarchical algorithm performs better initially but fails to converge to the optimal policy. A third algorithm which is given the optimal values for the bottleneck states at the start actually learns more slowly than both of the aforementioned algorithms. We give possible explanations for the observed behavior and suggestions for future work.

2 MDPs and Reinforcement Learning

A *Markov decision process (MDP)* models an agent acting in a stochastic environment with the aim of maximizing its expected long-term reward. The type of MDP we consider contains a finite set S of states, with s_0 being the start state. For each state $s \in S$, A_s is a finite set of actions available to the agent. P is the table of transition probabilities, where $P(s'|s,a)$ is the probability of a transition to state s' given that the agent performed action a in state s. R is the reward function, where $R(s,a)$ is the reward received by the agent given that it chose action a in state s.

A *policy* π is a mapping from states to actions. Solving an MDP amounts to finding a policy that maximizes the expected long-term reward. In this paper, we use the infinite-horizon discounted optimality criterion. Formally, the agent should maximize

$$E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))\right],$$

where $\gamma \in [0,1]$ is the discount factor. In order to model *episodic* tasks, we can include an absorbing state from which the agent can only receive an immediate reward of zero; a transition to the absorbing state corresponds to the end of an episode.

Algorithms for MDPs often solve for *value functions*. For a policy π, the state value function, $V^\pi(s)$, gives the expected total reward starting from state s and executing π. The state-action value function, $Q^\pi(s,a)$, gives the expected total reward starting from state s, executing action a, and executing π from then on.

When an explicit model is available, MDPs can be solved using standard dynamic programming techniques such as policy iteration or value iteration. When only a simulator or real experience are available, reinforcement learning methods are a reasonable choice. With these techniques, experience trajectories are used to learn a value function for a good policy. Actions taken on a trajectory are usually greedy with respect to the current value function, but *exploratory* actions must also be taken in order to discover better policies. One widely-used reinforcement learning algorithm is Q-learning [8], which updates the state-action value function after each transition from s to s' under

action a with the following rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right],$$

where α is called the learning rate.

3 Reinforcement Learning for Weakly-Coupled MDPs

Consider an MDP with a state set S that is partitioned into disjoint subsets $S_1, \ldots, S_m$. The *out-space* of a subset S_i, denoted $O(S_i)$, is defined to be the set of states not in S_i that are reachable in one step from S_i. The set of states $B = O(S_1) \cup \cdots \cup O(S_m)$ that belong to the out-space of at least one subset comprise the set of *bottleneck* states. If the set of bottleneck states is relatively small, we call the MDP *weakly-coupled*.

In [2], the authors describe an algorithm for weakly-coupled MDPs that can be described as a type of policy iteration. Initially, values for the bottleneck states are set arbitrarily. The low-level policy improvement phase involves solving each subproblem, treating the bottleneck state values as terminal rewards. The high-level policy evaluation phase consists of reevaluating the bottleneck states for these policies. Repeating these phases guarantees convergence to the optimal policy in a finite number of iterations.

The rules for backpropagating value information in our reinforcement learning algorithm are derived from the two phases mentioned above. Two benefits of our approach are that it doesn't require an explicit model and that learning can proceed simultaneously at the high level and at the low level.

We maintain two different value functions: a low-level state-action value function Q defined over all state-action pairs and a high-level state value function V_h defined over only bottleneck states. The low-level part of the learning is described as follows. Upon a transition to a non-bottleneck state, the standard Q-learning backup is applied. However, when a bottleneck state $s' \in B$ is encountered, the following backup rule is used:

$$Q(s,a) \leftarrow Q(s,a) + \alpha_l \left[R(s,a) + \gamma V_h(s') - Q(s,a) \right],$$

where α_l is a learning rate. For the purposes of learning, the bottleneck state is treated as a terminal state, and its value is the terminal reward. High-level backups occur only upon a transition to a bottleneck state. The backup rule is:

$$V_h(s) \leftarrow V_h(s) + \alpha_h [R + \gamma^k V_h(s') - V_h(s)],$$

where k denotes the number of time steps elapsed between the two bottleneck states, R is the cumulative discounted reward obtained over this time, and α_h is a learning rate.

It is possible to alternate between phases of low-level and high-level backups or to perform the backups simultaneously. Whether either approach converges to an optimal policy is an open problem. We chose the latter for our experiments because our preliminary work showed it be more promising.

4 Autonomous Planetary Rover Control

4.1 The Model

In this section we describe a simple version of the rover decision-making problem and how it fits within the weakly-coupled MDP framework. In our scenario, a rover is to operate autonomously for a period of time. It has an ordered sequence of sites along with priority information and estimated difficulty of obtaining data, and it must make decisions about which activities to perform and when to move from one site to the next. The goal is to maximize the amount of useful work that gets done during the time period.

The action set consists of taking a picture, performing a spectrometer experiment, and traversing to the next site in the sequence. Spectrometer experiments take more time and are more unpredictable than pictures, but they yield better data. The time to traverse between sites is a noisy function of the distance between the sites. The state features are the time remaining in the day, the current site number (from which priority and estimated difficulty are implicitly determined), the number of pictures taken at the current site, and whether or not satisfactory spectrometer data has been obtained at the current site. Formally, $S = T \times I \times P \times E$, where $T = \{0 \text{ min}, 5 \text{ min}, \ldots, 300 \text{ min}\}$ is the set of time values; $I = \{1, 2, 3, 4, 5\}$ is the set of sites; $P = \{0, 1, 2\}$ is the set of values for pictures taken; and $E = \{0, 1\}$ is the set of values for the quality of the spectrometer data. The start state is $s_0 = \langle 300, 1, 0, 0 \rangle$. The sequence of sites used for our experiments is shown in Table 1.

Table 1. The sequence of sites for the rover to investigate

Site	Priority	Estimated difficulty	Distance to next site
1	8	medium	3 m
2	5	hard	5 m
3	3	easy	7 m
4	2	easy	3 m
5	9	hard	N/A

A nonzero reward can only be obtained upon departure from a site and is a function of the site's priority and the data obtained at the site. The task is episodic with $\gamma = 1$. An episode ends when the time component reaches zero or the rover finishes investigating the last site. The aim is to find a policy that maximizes the expected total reward across all sites investigated during an episode. Because of limited time and nondeterministic action effects, the optimal action is not always obvious.

In order to see how this problem fits into the weakly-coupled MDP framework, consider the set of states resulting from a traversal between sites. In all of these states, the picture and spectrometer components of the state are reset to zero. The set $B = T \times I \times \{0\} \times \{0\}$ is taken to be the set of bottleneck states, and it is over this set that we define the high-level value function. Note that the bottleneck states comprise only 300 of the problem's 1,800 states.

4.2 Experiments

In our experiments, we tested Q-learning against the hierarchical algorithm on the problem mentioned in the previous section. In addition, we tested an algorithm that we call the *omniscient* hierarchical learning algorithm. This algorithm is the same as the hierarchical algorithm, except that the values for the bottleneck states are fixed to optimal from the start, and only low-level backups are performed. By fixing the bottleneck values, the problem is completely decomposed from the start. Of course, this cannot be done in practice, but it is interesting for the purpose of comparison.

For the experiments, all values were initialized to zero, and we used ϵ-greedy exploration with $\epsilon = 0.1$ [6]. For the results shown, all of the learning rates were set to 0.1 (we obtained qualitatively similar results with learning rates of 0.01, 0.05, and 0.2). Figure 1 shows the total reward per episode plotted against the number of episodes of learning. The points on the curves represent averages over periods 1000 episodes.

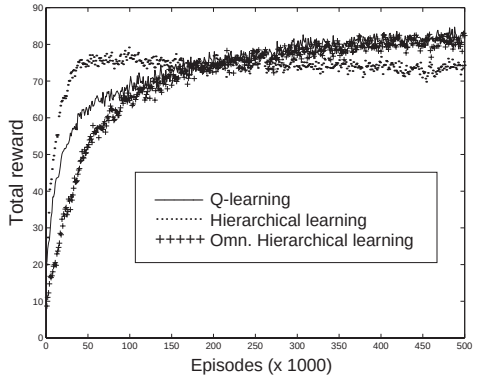


Fig. 1. Learning curves for Q-learning, hierarchical learning, and omniscient hierarchical learning

A somewhat counterintuitive result is that the omniscient hierarchical algorithm performs worse than both the original hierarchical algorithm and Q-learning during the early stages. One factor contributing to this is the initialization of the state-action values to zero. During the early episodes of learning, the value of the "leave" action grows more quickly than the values for the other actions because it is the only one that leads directly to a highly-valued bottleneck state. Thus the agent frequently leaves a site without having gathered any data. This result demonstrates that decomposability doesn't always guarantee a more efficient solution.

The second result to note is that the hierarchical algorithm performs better than Q-learning initially, but then fails to converge to the optimal policy. It is intuitively plausible that the hierarchical algorithm should go faster, since it implicitly forms an abstract process involving bottleneck states and propagates value information over multiple time steps. It also makes sense that the algorithm doesn't converge once we consider that the high-level backups are *off policy*. This means that bottleneck states are evaluated for the policy that is being executed, and this policy always includes non-greedy exploratory

actions. Algorithms such as Q-learning, on the other hand, learn about the policy that is greedy with respect to the value function regardless of which policy is actually being executed.

5 Conclusion

We studied a hierarchical reinforcement learning algorithm for weakly-coupled MDPs, using a problem in planetary rover control as a testbed. Our results indicate that the decomposability of these problems can lead to greater efficiency in learning, but the conditions under which this will happen are not yet well understood. Perhaps experimentation with different low-level and high-level learning rates could shed some insight. Also, experimental results from other weakly-coupled MDPs besides the rover problem would be valuable. Finally, a more detailed theoretical investigation may yield an algorithm similar to ours that is provably convergent.

On the application side, we plan to develop a more realistic and complex simulator of the rover decision-making problem. In this simulator, the rover will choose among multiple sites to traverse to. It will also have to manage its data storage and battery capacity and perform activities during constrained time intervals. The state space of the model will most likely be too large to explicitly store a value for each state. We will instead have to use some form of function approximation.

Acknowledgements

The authors thank Rich Washington, John Bresina, Balaraman Ravindran, and Ted Perkins for helpful conversations. This work was supported in part by the NSF under grants IRI-9624992 and IIS-9907331, and by NASA under grants NAG-2-1394 and NAG-2-1463. Daniel Bernstein was supported by an NSF Graduate Research Fellowship and a NASA SSRP Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF or NASA.

References

1. Boutilier, C., Dean, T. & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. Journal of Artificial Intelligence Research, 1, 1–93.
2. Dean, T. & Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. In IJCAI-95.
3. Foreister, J.-P. & Varaiya, P. (1978). Multilayer control of large Markov chains. IEEE Transactions on Automatic Control, 23(2), 298–304.
4. Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T. & Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. In UAI-98.
5. Parr, R. (1998). Flexible decomposition algorithms for weakly coupled Markov decision problems. In UAI-98.
6. Sutton, R. S. & Barto, A. G. (1998). Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press.
7. Sutton, R. S., Precup, D. & Singh, S. (2000). Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. Artificial Intelligence, 112, 181–211.
8. Watkins, C. (1989). Learning from Delayed Rewards. PhD thesis, Cambridge University, Cambridge, England.

Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space

Piergiorgio Bertoli[1], Alessandro Cimatti[1], Marco Roveri[1,2]
{bertoli,cimatti,roveri}@irst.itc.it
[1] ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy
[2] DSI, University of Milano, Via Comelico 39, 20135 Milano, Italy

Abstract. Planning under partial observability in nondeterministic domains is a very significant and challenging problem, which requires dealing with uncertainty together with and-or search. In this paper, we propose a new algorithm for tackling this problem, able to generate conditional plans that are guaranteed to achieve the goal despite of the uncertainty in the initial condition and the uncertain effects of actions. The proposed algorithm combines heuristic search in the and-or space of beliefs with symbolic BDD-based techniques, and is fully amenable to the use of selection functions. The experimental evaluation shows that heuristic-symbolic search may behave much better than state-of-the-art search algorithms, based on a depth-first search (DFS) style, on several domains.

1 Introduction

In this paper, we tackle the problem of conditional planning under partial observability, where only part of the information concerning the domain status is available at run time. In its generality, this problem is extremely challenging. Compared to the limit case of full observability, it requires dealing with uncertainty about the state in which the actions will be executed. Compared to the limit case of null observability, also known as conformant planning, it requires the ability to search for, and construct, plans representing conditional courses of actions. Several approaches to this problem have been previously proposed, e.g. [WAS98], based on extensions of GraphPlan, and [BG00], based on Partially Observable Markov Decision Processes (POMDP).

Our work builds on the approach proposed in [BCRT01], where planning is seen as and-or search of the (possibly cyclic) graph induced by the domain, and BDD-based techniques borrowed from symbolic model checking provide efficient search primitives. We propose a new algorithm for planning under partial observability, able to generate conditional acyclic plans that are guaranteed to achieve the goal despite of the uncertainty in the initial condition and in the effects of actions. The main feature of the algorithm is the heuristic style of the search, that is amenable to the use of selection functions, and is fully compatible with the use of a symbolic, BDD-based representation. We call this approach *heuristic-symbolic* search. The proposed approach differs from (and improves) the depth-first search proposed in [BCRT01] in several respects. First, depending on the selection function, it can implement different styles of search, including DFS. Furthermore, the use of selection functions allows to overcome potentially bad initial choices, and can therefore result in more efficient computations and higher quality plans. Finally, it opens up the possibility of using preprocessing techniques for determining domain/problem-dependent heuristics. The heuristic-symbolic algorithm was implemented in the MBP planner [BCP+01], and an extensive experimental evaluation was carried out. The results show that, even considering a simple domain-independent heuristic, for several classes of problems the heuristic-symbolic algorithm significantly improves the performance and constructs better plans with respect to DFS.

The paper is organized as follows. In Section 2 we describe partially observable planning domains and conditional planning. In Section 3 we present the planning algorithm. In Section 4 we give an overview of the experimental evaluation, draw some conclusions and discuss some future work.

2 Domains, Plans, and Planning Problems

A partially observable planning domain is a tuple $\langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{O}, \mathcal{X} \rangle$. $\mathcal{P}$ is a finite set of propositions. $\mathcal{S} \subseteq Pow(\mathcal{P})$ is the set of states. $\mathcal{A}$ is a finite set of actions. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation, describing the effects of (possibly nondeterministic) action execution. We say that an action a is applicable in a state s iff there exists at least one state s' such that $\mathcal{R}(s, a, s')$. $\mathcal{O}$ is a set of boolean observation variables, whose values can be observed during plan execution. $\mathcal{X} : \mathcal{O} \rightarrow Pow(\mathcal{S} \times \{\top, \bot\})$ is the observation relation. Intuitively, $\mathcal{X}$ associates each possible value of each observation variable to the set of states where observing the variable returns such value. (We consider observation variables to be always defined, and independent from actions performed prior to observing. The full framework (see [BCRT01]) and the actual MBP implementation are free from these constraints. The current presentation is simplified for reasons of space.)

We consider conditional plans, branching on the value of observable variables. A plan for a domain $\mathcal{D}$ is either the empty plan ϵ, an action $a \in \mathcal{A}$, the concatenation $\pi_1; \pi_2$ of two plans π_1 and π_2, or the conditional plan $o \ ? \ \pi_1 \ : \ \pi_2$ (read "if o then π_1 else π_2"), with $o \in \mathcal{O}$. The execution of a plan π must take into account, at each step, that the executor can be unable to distinguish between a set of possible states for the current situation. We call such a set of indistinguishable states a "belief state". An action a is applicable to a belief state Bs iff a is applicable in all states of Bs. Intuitively, the execution of a conditional plan π starting from a belief state Bs results into a set of states recursively defined over the structure of π by either (a) the application of the transition relation, if an action is performed, or (b) the union of the executions on every branch resulting from the possible observation values for a variable o, if the plan branches on o. We say that a plan is applicable on Bs if no non-applicable actions are met during its execution on Bs. A planning problem is a 3-tuple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ whose components are a planning domain $\mathcal{D}$, a set of initial states $\mathcal{I}$ and a set of goal states $\mathcal{G}$. A solution consists of a plan π which is applicable over $\mathcal{I}$, and whose execution on $\mathcal{I}$ results in a belief state equal to $\mathcal{G}$ or contained into it.

Consider the example of a simple robot navigation domain $\mathcal{D}_0$, a 2x2 room with an extra wall (Figure 1). The propositions of the domain are `NW`, `NE`, `SW`, and `SE`, i.e. the four positions in the room. Exactly one of them holds in each of the four states in $\mathcal{S}$. The robot can move in the four directions, unless the action is not applicable due to a wall standing in the direction of motion. At each time tick, the information of walls proximity in each direction is available to the robot (observation variables `WallN`, `WallS`, `WallW` and `WallE`). In the figure, the execution of the plan π_0 = `GoEast ; WallN ? (GoSouth ; GoWest ) : GoWest`, starting from the uncertain initial condition `NW` or `SW`, is outlined by solid lines. The plan π_0 is a solution for the problem P_0 = {$\mathcal{D}_0$, {`NW`, `SW`}, {`SW`} }.

3 Planning under Partial Observability

When planning under partial observability, the search space can be seen as an and-or graph, recursively constructed from the initial belief state, expanding each encountered belief state by every possible combination of applicable actions and observations. The graph is possibly cyclic; in order to rule out cyclic behaviors, however, the exploration

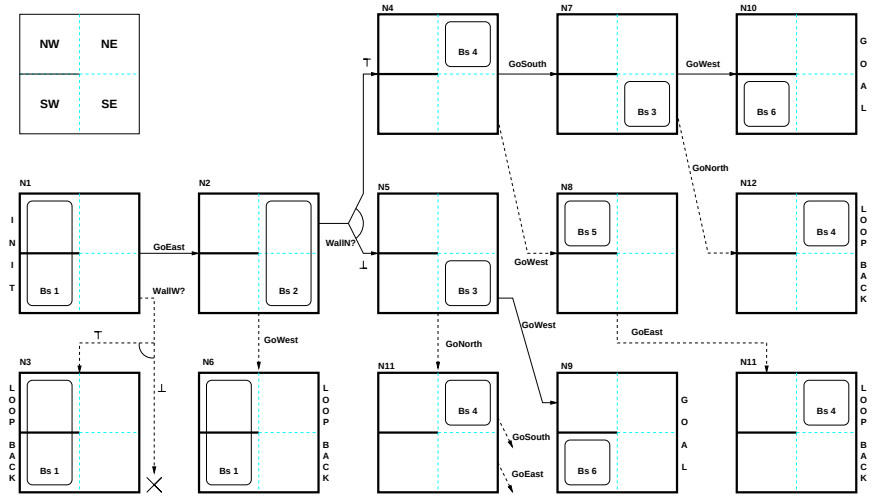


Fig. 1. A simple robot navigation domain

can be limited to its acyclic prefix. Figure 1 depicts the finite prefix of the search space for the problem P_0 described above. Each node in the prefix is associated to a path, describing how the node has been reached, and to the corresponding belief state.

The prefix is constructed by expanding each node in all possible ways, each represented by an outgoing arc. Single-outcome arcs correspond to applicable actions (action execution is deterministic in belief space). For instance, N4 expands into N7 and N8. Multiple outcome arcs correspond to observations. For instance, node N2 results in nodes N4 and N5, corresponding to the observation of `WallN`. The application of an observation is what gives the "and" component in the search space: we have a solution for (the belief state associated with) N2 if we have a solution for both N4 and N5 (whose associated belief states are obtained by conjoining the beliefs associated by $\mathcal{X}$ to the observed values of `WallN` with the belief state associated to N2). Some non-informative observations are not reported in the graph.

The expansion of a node n is halted when (a) n is associated with a belief state contained in the goal, or (b) n is a loop back, i.e. it has an ancestor node with the same belief state. For instance, node N6 loops back onto node N1, while node N11 loops back onto node N4. Node N9 and N10 are associated with the goal belief state.

The planning algorithm for conditional planning under partial observability is described in Figure 2. It takes in input the initial belief state and the goal belief state, while the domain representation is assumed to be globally available to the subroutines. The algorithm incrementally constructs the finite acyclic prefix described above. The algorithm relies on an extended data structure, stored in the $graph$ variable, which represents the prefix being constructed. Each node in the structure is associated with a belief state and a path. In addition to son-ancestor links, the graph has links between the nodes in the equivalence classes induced by equality on associated belief states, and presents an explicit representation of the frontier of nodes to be expanded. A success pool contains the solved nodes for the graph.

At line 1, the algorithm initializes the graph: the root node corresponds to the initial belief state, and the success pool contains the goal. Then (lines 2-24) the iteration pro-

```
HEURSYMCONDPLAN(I, G)
1   graph := MKINITIALGRAPH(I, G);
2   while (GRAPHROOTMARK(graph) ∉ {Success, Failure})
3       node := EXTRACTNODEFROMFRONTIER(graph);
4       if (SUCCESSPOOLYIELDSSUCCESS(node, graph))
5           MARKNODEASSUCCESS(node);
6           NODESETPLAN(node,RETRIEVEPLAN(node, graph));
7           PROPAGATESUCCESSONTREE(node,graph);
8           PROPAGATESUCCESSONEQCLASS(node,graph);
9       else
10          orexp := EXPANDNODEWITHACTIONS(node);
11          andexp := EXPANDNODEWITHOBSERVATIONS(node);
12          EXTENDGRAPHOR(orexp, node, graph);
13          EXTENDGRAPHAND(andexp, node, graph);
14          if (SONSYIELDSUCCESS(node))
15              MARKNODEASSUCCESS(node);
16              NODESETPLAN(node,BUILDPLAN(node));
17              PROPAGATESUCCESSONTREE(node, graph);
18              PROPAGATESUCCESSONEQCLASS(node, graph);
19          else if (SONSYIELDFAILURE(node))
20              MARKNODEASFAILURE(node, graph);
21              NODEBUILDFAILUREREASON(node, graph);
22              PROPAGATEFAILUREONTREE(node, graph);
23              PROPAGATEFAILUREONEQCLASS(node, graph);
24  end while
25  if (GRAPHROOTMARK(graph) = Success)
26      return GRAPHROOTPLAN(graph);
27  else
28      return Failure;
```

Fig. 2. The planning algorithm

ceeds by selecting a node and expanding it, until a solution is found or the absence of a solution is detected. At loop exit, either a plan or a failure is returned (lines 25-28).

With the first step in the loop, at line 3, a node is extracted from the frontier in order to be expanded. The EXTRACTNODEFROMFRONTIER primitive embodies the selection criterion and is responsible for the style (and the effectiveness) of the search. Then, at line 4, we check whether the belief state associated to the selected $node$ is entailed by some previously solved belief state in the success pool. If so, the formerly detected plan is reused for $node$, which is marked as success. Moreover, the success is recursively propagated both to the ancestors of $node$ (line 7) and to the nodes in its equivalence class (line 8). The rules for success propagation directly derive from the and-or graph semantics. Recursive success propagation takes also care of removing descendents of success nodes from the frontier (as their expansion would be useless).

If the success of $node$ cannot be derived by the success pool, then the expansion of $node$ is attempted, computing the nodes resulting from possible actions (line 10) and observations (line 11). The graph extension steps, at lines 12-13, construct the nodes associated to the expansion, and add them to the graph, also doing the bookkeeping operations needed to update the frontier and the links between nodes. In particular, for each node, the associated status is computed. For instance, if a newly constructed node has a belief state that is already associated with a plan, then the node is marked

as success. Newly constructed nodes are also checked for loops, i.e. if they have an ancestor node with the same belief state then they are marked as failure.

If it is possible to state the success of $node$ based on the status of the newly introduced sons (primitive SONSYIELDSUCCESS at line 14), then the same operations at line 5-8 for success propagation are executed. Similarly, at lines 19-23, if it is possible to state the failure of $node$ based on the status of the newly introduced sons, failure is propagated throughout the and-or graph. Failure can happen, for instance, due to loop detection. The failure of the node is stored in such a way that it can be reused in the following search attempts. Notice however that, differently from success, a failure depends on the node path. For instance, in a subsequent search attempt it could be possible to reach a belief state with a non-cyclic path. Therefore, each belief state is associated with a set of belief states representing the failure reason. Intuitively, the failure reason contains the sets of belief states that caused a loop in all the search attempts originating from the belief state marked with failure.

The algorithm is integrated in MBP [BCP+01], a general planner for nondeterministic domains which allows for conditional planning under full observability, also considering temporally extended goals, and for conformant planning. MBP is based on the use of symbolic model checking techniques [McM93]. In particular, it relies on Binary Decision Diagrams, structures that allow for a compact representation of sets of states and an efficient expansion of the search space (see [CR00] for an introduction to the use of BDDs in planning).

4 Results and Conclusions

We carried out an extensive experimental analysis of the heuristic-symbolic algorithm presented in previous section, comparing it with the DFS approach of [BCRT01], shown to outperform other conditional planners such as SGP and GPT. For lack of space, we only provide a high-level description of the considered domains and results. The details can be found in [BCR01]. We considered the standard benchmark problems for PO planning used in [BCRT01]: MAZE, Empty Room (ER), RING. In the MAZE, a moving robot must reach a fixed position in a maze, starting from anywhere and being able to observe the walls around its current position. Basically, this problem reduces to gathering knowledge about the robot position. Unless the maze is significantly symmetric, almost at each move of the robot, observing contributes to the purpose. Furthermore, the problem is highly constrained: observing is forced in many situations by the lack of applicable actions prior to that. In the ER, the same problem is tackled considering, rather than a maze, a wide empty room; the robot starts from anywhere in the room. In this formulation, most moves of the robot will lead to "enabling" some useful sensing. In the RING, the aim is to have all windows of a ring of connected rooms locked by a moving robot. Each window must be closed, if open, before being locked (if unlocked). Here, the key issue is that of locality: before moving around, the robot should better solve the local problem of locking the local window. Otherwise, plans may become extremely lengthy. Thus, observing and moving must be interleaved "in a sensible way". Furthermore, we considered some variations to the ER. In the VER problem, the robot initially is in one of two positions near the center of the room. This forces the robot to execute long sequences of actions before being able to gather some useful information from sensing the walls. In the ERS, a portion of the empty room is a "sink", i.e. once entered there, the robot cannot exit it.

The performance of the heuristic-symbolic algorithm heavily depends on the selection function, that controls which portions of the search space are explored. The

problem of finding an effective, problem dependent selection function for controlling and-or search appears to be in general very hard. In all our experiments, we considered a simple structural selection function, that gives high scores to nodes whose equivalence class contains many open nodes and few failed ones. In spite of this choice, the results are quite promising. In the RING, ER, VER, ERS problems the timing of the search, and the length of the plan (defined as its maximum depth) are much better than in the original DFS search. The MAZE problem evidences a reasonable loss of efficiency. This is due to the fact that the problem is very constrained, and drives the DFS search accordingly. In this case, the overhead of maintaining a graph structure and having explicit propagation routines explains the result.

The conclusion that can be drawn is that giving up the DFS-style search is in general an advantage, leading in many cases to better results even in absence of highly tuned scoring mechanisms, and opens up the possibility for further improvements. Future research will be directed to the definition of preprocessing techniques and more effective heuristic functions, with the goal to obtain "smarter" behaviors from the heuristic-symbolic algorithm. Another direction of future research is the extension of the partially observable approach presented in this paper to find strong cyclic solutions, and to deal with goals expressed in a temporal logic.

References

[BCP+01] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a Model Based Planner. In *Proc. of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, Seattle, August 2001.

[BCR01] P. Bertoli, A. Cimatti, and M. Roveri. Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space. Technical report, IRST, Trento, Italy, July 2001. Extended version of this paper.

[BCRT01] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc.* 7^{th} *International Joint Conference on Artificial Intelligence (IJCAI-01)*. AAAI Press, August 2001.

[BG00] B. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In S. Chien, S. Kambhampati, and C.A. Knoblock, editors, 5^{th} *International Conference on Artificial Intelligence Planning and Scheduling*, pages 52–61. AAAI-Press, April 2000.

[CR00] A. Cimatti and M. Roveri. Conformant Planning via Symbolic Model Checking. *Journal of Artificial Intelligence Research (JAIR)*, 13:305–338, 2000.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[WAS98] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 897–904, Menlo Park, July 26–30 1998. AAAI Press.

Beyond Plan Length: Heuristic Search Planning for Maximum Reward Problems.

Jason Farquhar and Chris Harris

Image Speech and Intelligent Systems
Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
[jdrf99r;cjh]@ecs.soton.ac.uk

Abstract. Recently automatic extraction of heuristic estimates has been shown to be extremely fruitful when applied to classical planning domains. We present a simple extension to the heuristic extraction process from the well-known HSP system that allows us to apply it to reward maximisation problems. This extension involves computing an estimate of the maximal reward obtainable from a given state by ignoring delete lists. We also describe how to improve the accuracy of this estimate using any available mutual exclusion information. In this way we seek to apply recent advances in classical planning to a broader range of problems.

Keywords: Domain Independent Planning, Reward Based Planning, Heuristic Search Planners.

1 Introduction

In this paper we investigate reward maximisation as an alternative to plan length for the optimisation criteria in STRIPS style problems. In reward maximisation problems we attempt to maximise the total reward obtained from the states visited and actions taken during plan execution, where the reward is an aribitary real-valued function over states and actions. In particular we focus on reward problems where the planners objectives are specified through the rewards allocated to different world states rather than as an explicit goal which *must* be achieved.

Inspired by the success of heuristic search in efficiently solving goal-based STRIPS problems (Bac00; BG99; HN01) we suggest that similar methods may be used in reward maximisation problems. To investigate this idea we present a modification of the heuristic used in HSP which is applicable in the reward maximisation case.

This paper is organised as follows. Section 2 presents a mathematical model of STRIPS problems and its reward based extensions. Section 3 gives the derivation of our new heuristic and an outline of the algorithm used to calculate it. The final sections discuss future work (Sec 4), compare related work (Sec 5) and present conclusions (Sec 6).

2 Reward Based Planning

Following (BG99) we represent a conventional STRIPS (FN71) domain as a tuple $D = \langle A, O \rangle$, where A is a set of atoms and O is a set of operators. The operators $op \in O$ and atoms $a \in A$ are all assumed ground (all variables replaced by constants). Each operator

has precondition, add and delete lists which we denote as Pre(op), Add(op) and Del(op) respectively, given by sets of atoms from A.

Such a domain can be seen as representing a state space where:

1. the states $s \in S$ are finite sets of atoms from A
2. the state transition function $f(s, op)$ which maps from states to states is given by:

$$s' = f(s, op) = \begin{cases} s \cup \text{Add}(op) \setminus \text{Del}(op) & \text{if Pre}(op) \subseteq s \\ \text{undefined} & \text{otherwise.} \end{cases} \quad (1)$$

3. the result of applying a sequence of operators is defined recursively as

$$s_n = f(s_0, \langle op_1, .., op_n \rangle) = f(f(s_0, \langle op_1, ..., op_{n-1} \rangle), op_n) \quad (2)$$

In reward based planning the domain description is augmented to give $\mathcal{P} = \langle A, O, I, R \rangle$ where $I \subset A$ represents the initial situation and R is the reward function which maps from situations to real valued rewards. R consists of two components, a state based component, $R : s \mapsto \mathbb{R}$, and an operator based component, $R : op \mapsto \mathbb{R}$. The solution to a reward based planning problem is a sequence of operators $P = \langle op_1, ..., op_n \rangle$ that *maximises* the total reward received from the states visited and operators applied during plan execution.

One particular problem with reward based planning not found in goal based planning is the possibility of cyclic solutions with infinite reward. Such infinities are difficult to work with so it is usual to modify the optimisation criteria to remove them; for example, by discounting future rewards (Put94), optimising with respect to reward rate, or optimising with respect to a finite planning horizon. A finite horizon is used in this work though the method could be applied in the other cases .

3 Heuristics for Reward Based Planning Problems

Heuristic search planners use a heuristic function $h(s)$ to guide solution search in state space. To develop an effective heuristic we use the same trick that proved so effective in STRIPS planning (BG99; HN01), i.e. we solve a relaxed problem ignoring operator delete lists and use this solution as a heuristic estimate in the original problem. Unfortunately solving even the relaxed problem can be shown to be NP-hard (BG99) so an approximate solution must be used.

In STRIPS problems one of the most successful approximations is that used in the HSP system developed by Bonet and Genffer (BG99). This decomposes the problem of finding the shortest path to the goal state into one of finding the shortest path to each individual atom from which an estimate of the goal distance is reconstructed. This decomposition and reconstruction is performed because the number of atoms, $|A|$ is generally *much* smaller than the number of states, i.e. subsets of atoms, $|S|$, (which is exponential in the number of atoms, $|S| \leq |2^{|A|}|$). Hence performing computations in atom space can be significantly cheaper in time and space than the equivalent computations in state space.

For reward based problems we propose to use a modifed version of the same approximation technique of decomposing and resconstructing state values from atom values. We begin by defining the *value*, $V(s, t)$, of state s as the maximal reward obtainable in getting

from the initial state I to this state in t steps. This value could be calculated directly in state space using the forward Bellman Equation:

$$V(s,t) = R(s) + \max_{op \in O} \left[R(op) + V(f^{-1}(s,op), t-1) \right] \tag{3}$$

where $V(s,t)$ is the value of the state s at time step t, $R(s)$ and $R(op)$ are the rewards for being in state s and performing operation op respectively, $f^{-1}(s,op)$ is the inverse state transition operator which returns the state s' from which application of operator op results in the new state s, and $V(I,0) = 0$.

The problem defined by (3) is equivalent to finding a maximal weight path through a weighted directed graph. The nodes represent states, edges the operators, and the edge and node weights the operator and state rewards respectively. A number of efficient algorithms which are polynomial in $|S|$ can be used to solve this problem. Unfortunately as mentioned above, $|S|$ is generally exponential in the number of atoms making even these algorithms costly. Hence we approximate (3) by re-formulating the problem to apply over the smaller space of atoms. This gives the equations (4) and (5).

$$V(p,t) = \max_{p \in \mathrm{Pre}(r)} R(r,t) + \max_{p \in \mathrm{Eff}(op)} \left(R(op) + V(\{\mathrm{Pre}(op)\}, t-1) \right) \tag{4}$$

$$V(p,0) = \begin{cases} 0 & \text{if } p \in I \\ \text{undefined} & \text{otherwise} \end{cases} \tag{5}$$

where $V(\mathrm{Pre}(op), t)$ is the reward for for being in atom set $\{p : p \in \mathrm{Pre}(op)\}$ at time step t, and $\mathrm{Pre}(r)$ is the set of atoms which define reward state r. $R(r,t)$ is the reward obtained from being in reward state r at time t and is equal to the value of the reward state $R(r)$ if all the atoms in r are valid at time t and undefined otherwise.

Equation (4) defines the estimated value of an atom at time step t is the sum of the immediate reward received due to the current state, $R(r,t)$, and the propagated total reward of the maximum reward path to this atom from the initial state. Equation (5) sets the initial value of the atoms.

The accuracy of the function, $V(\{B\}, t)$, used to estimate value of an atom set, B, from the values of its consistituent atoms, $p \subset B$, is critical to the accuracy of the heuristic and hence performance of the planner. In (BG99) Bonnet and Genffer suggest using either the sum or maximum of the atom values. Using the sum pessimistically assumes that each atom is totally independent, hence the shortest path to the set becomes the sum of the best paths to each atom in the set. Using the maximum optimistically assumes that the atoms are totally dependent such that any path which achieves one atom will also achieve all other atoms which can be reached with shorter paths. Hence the shortest path to an atom set becomes the length of the shortest path to the last atom achieved.

In the reward maximisation case things become a little more complex. If independence is assumed then the atom set can only be valid when t is greater than the sum of the initial values for each atom in the set. If total dependence is assumed then the value of the set becomes the value of the highest reward path which could achieve the set, i.e. the last atom achieved initially and after that the highest reward path longer than the sets initially valid length. In this case, which is used in this paper, we obtain Equation (6).

$$V(B,t) = \begin{cases} \max_{a \in B, l \leq \text{len}(a) \leq t} V(a,t) & \text{if } \forall a \in B,\ V(a,t) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases} \quad (6)$$

where B is the set of atoms, $\text{len}(a)$ is the number of operations required to obtain atom a's value $V(a,t)$ and l is the number of operations required to first make B valid.

Solving the equations (4,6,5) also corresponds to finding the maximum weight path in a graph, the *connectivity graph* (HN01). In this graph atoms are nodes and operators/rewards are high order edges which only become available after a certain subset of the nodes (their preconditions) are valid.

Computation of the atom values can be done in polynomial time using a GraphPlan like forward propagation procedure based upon the Connectivity Graph. Briefly, the algorithm proceeds in a time step by time step manner propagating tokens from atoms to operators and rewards. The operators/rewards are then identified as available for propagating reward and atom validity to their effects when all their pre-conditions are valid. The set of valid atoms is then used to compute the updated value for all the atoms using equations (4,6,5).

Using the value function computed in this way, the heuristic value of the state s in the original problem is defined as the maximal value of a valid atom in the final layer of the relaxed graph, Eqn (7).

$$h(s,t) \stackrel{def}{=} \max_{p \in P^t} V(p,t) \quad (7)$$

where t is the maximum time step to which the relaxed solution has been computed and P^t is the set of atoms valid, i.e. with defined values, at time t.

3.1 Improving the Estimate Using Mutual Exclusion Information

One problem with the heuristic estimates produced by the above procedure is that it takes no account of the negative interactions between atoms. This is caused by ignoring the operators delete lists allowing extra paths to states or atoms being possible in the relaxed problem which do not exist in the original problem, making states become valid earlier or with higher reward. This is the same problem of ignored negative interactions examined in (NNK00; NK00) and can be addressed in a similar way using any available mutual exclusion information.

Mutual exclusions, called mutexs from now on, are used extensively in the Graphplan algorithm (BF97) and represent pairs of atoms that cannot occur together at some depth in any plan. In the heuristic value computation this information can be used to close off some of the extra paths by preventing or delay operators/rewards from becoming valid until their pre-conditions are all non mutex. For negligible cost this information can be used in the value computation algorithm by annotating each operator/reward by the first plan depth at which it is applicable, i.e. all its pre-conditions are non mutex. Then the operators/reward are restricted to only be available after this depth. The technique can be used both for *static* mutexs which hold for all states and plan lengths and *dynamic* mutexs which hold only up to a certain plan length.

The cost of computing the additional mutexs can be controlled by only calculating the static mutex's once at the start of problem solving, for example by running GraphPlans

full graph construction algorithm to *level-off* from the initial state. Any mutex's which hold at *level-off* this point are static. Additional dynamic mutex's and then be computed to any required degree of accuracy only when necessary.

4 Further Work

A prototype reward based planner has been implemented using the above heuristic evaluation function both with and without the mutal exclusion enhanchements. Initial results appear to validate the system with the heuristic values showing good correlation to the true value of a state. We are currently in the process of developing an A* search engine a full test suite for the planner. We then intend to perform comparisons with other planners in both conventional STRIPS and reward based domains.

5 Related Work

There are obviously rich connections between this work and existing work on Graphplan (BF97) and the heuristic state space search planners (BG99; HN01) upon which it is based. The idea of using mutual exclusion information to account for negative interactions and hence improve the quality of the heuristic estimates is similar to that used in (NNK00) to improve the heuristic estimates in regression search.

This work is also closely related to work on adding probabilistic (BL98) and decision theoretic (PC00) abilities to the Graphplan algorithm. These systems rely on propagating additional probability information through the planning graph in much the same way that rewards are propagated in this work. This work also has significant connections to work on decision theoretic planning, where reward based formulations are also used. Traditionally these systems have used dynamic programming over a graphical representation of state space to find optimal solutions (Put94). As discussed above in Sec 2 and in (BDH99) this works well for reasonable state space sizes but tends to become infeasible for very large state spaces. Use of heuristic search to address such large problems has recently been proposed by Bonet and Geffner (BG00).

6 Conclusions

A method for extending the techniques of heuristic planning, as used in the well known HSP system, to the more expressive language of reward based planning was presented. The development of a domain independent heuristic for reward maximisation problems forms the crux of our work. This heuristic is based upon computing an estimate for the maximal reward obtainable in a relaxed problem where delete lists are ignored. We have shown how our heuristic function was developed to cope with reward accumulation and goalless planning problems. We have also demonstrated how this heuristic estimate can be improved by using any available mutual exclusion information to take some account of negative interactions.

References

[Bac00] F. Bacchus. Results of the aips 2000 planning competition, 2000. URL: http://www.cs.tronto.edu/aips-2000.

[BDH99] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence*, 11:1–94, 1999.

[BF97] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[BG99] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of ECP-99*. Springer, 1999.

[BG00] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc AAAI-2000*, 2000.

[BL98] Avrim L. Blum and John C. Langford. Probabilistic planning in the graphplan framework. Technical report, Carnegie Mellon University, School of Computer Science, CMU, Pittsburgh, PA 15213-3891, 1998.

[FN71] R. E. Fikes and N. J. Nilsson. Strips: a new approach to the application of theorm proving to problem solving. *Artificial Intelligence*, 2(3-4):189–203, 1971.

[HN01] Jorg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[NK00] XuanLong Nguyen and Subbarao Kambhampati. Extracting effective and admissible state space search heuristics for the planning graph. Technical report, Arizona State University, Dept. of Computer Science and Engineering, Tempe, AZ 85287-5406, 2000.

[NNK00] R.S. Nigenda, X. Nguyen, and S. Kambhampati. Altalt: Combining the advantages of graphplan and heuristic state space search. Technical report, Arizona State University, Dept. of Computer Science and Engineering, Tempe, AZ 85287-5406, 2000.

[PC00] G. Peterson and D. J. Cook. Decision-theoretic planning in the graphplan framework. In *Proc AAAI-2000*, 2000.

[Put94] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

Combining Two Fast-Learning Real-Time Search Algorithms Yields Even Faster Learning*

David Furcy and Sven Koenig

Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280
{dfurcy, skoenig}@cc.gatech.edu

Abstract. Real-time search methods, such as LRTA*, have been used to solve a wide variety of planning problems because they can make decisions fast and still converge to a minimum-cost plan if they solve the same planning task repeatedly. In this paper, we perform an empirical evaluation of two existing variants of LRTA* that were developed to speed up its convergence, namely HLRTA* and FALCONS. Our experimental results demonstrate that these two real-time search methods have complementary strengths and can be combined. We call the new real-time search method eFALCONS and show that it converges with fewer actions to a minimum-cost plan than LRTA*, HLRTA*, and FALCONS.

1 Introduction

Real-time (heuristic) search methods have been used to solve a wide variety of planning problems. Learning Real-Time A* (LRTA*) [7] is probably the best-known real-time search method. Unlike traditional search methods it can not only act in real-time but also amortize learning over several planning episodes if it solves the same planning task repeatedly. This allows it to find a suboptimal plan fast and then improve the plan until it follows a minimum-cost plan. Researchers have recently attempted to speed up its convergence while maintaining its advantages over traditional search methods, that is, without increasing its lookahead. Ishida and Shimbo, for example, developed ε-search to speed up the convergence of LRTA* by sacrificing the optimality of the resulting plan [5, 6]. In this paper, on the other hand, we study real-time search methods that speed up the convergence of LRTA* without sacrificing optimality, namely HLRTA* [8] (which is similar to SLRTA* [1]) and our own FALCONS [2]. We present the first thorough empirical evaluation of HLRTA* and show that it and FALCONS have complementary strengths that can be combined. We call the resulting real-time

* We thank Stefan Edelkamp for introducing us to HLRTA*, Richard Korf for making Thorpe's thesis about HLRTA* available to us, and James Irizarry for re-implementing HLRTA*. The Intelligent Decision-Making Group is partly supported by NSF awards to Sven Koenig under contracts IIS-9984827 and IIS-0098807 as well as an IBM faculty partnership award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

Table 1. Comparison of HLRTA* and FALCONS with LRTA*

Performance Measure	Average Over	Speedup Over LRTA* HLRTA*	FALCONS
Number of Actions to Convergence	All 15 Cases	2.08%	18.84%
	7 Most Informed Cases	-0.69%	28.73%
	7 Least Informed Cases	4.50%	8.86%
Number of Trials to Convergence	All 15 Cases	-11.06%	42.33%
	7 Most Informed Cases	-16.68%	43.87%
	7 Least Informed Cases	-5.17%	40.52%
Number of Actions in First Trial	All 15 Cases	7.04%	-21.90%
	7 Most Informed Cases	11.65%	-49.81%
	7 Least Informed Cases	-0.99%	-0.14%

search method Even FAster Learning and CONverging Search (eFALCONS) and show that it converges with fewer actions to a minimum-cost plan than LRTA*, HLRTA*, and FALCONS, even though it looks at the same states when it selects successors on undirected graphs and is not more knowledge-intensive to implement.

2 Motivation for Combining HLRTA* and FALCONS

HLRTA* keeps the successor-selection rule of LRTA* but improves its value-update rule, while FALCONS keeps the value-update rule of LRTA* but improves its successor-selection rule. In the following, we compare these two real-time search methods with LRTA* averaged over 1000 runs in seven domains with two or three different heuristic functions each, for a total of fifteen distinct experimental cases that we have previously used in [3]. As required by the real-time search methods, all domains are finite, all of their states have finite goal distances, and all heuristic functions do not overestimate the true distances. We use three different performance measures. The main performance measure is the number of actions until convergence (that is, until the real-time search methods repeatedly execute a minimum-cost plan). The other performance measures are the number of trials to convergence and the number of actions in the first trial, where a trial consists of all actions until the goal is reached and the real-time search method is reset to the start. Table 1 summarizes our empirical results. The number of actions to convergence of FALCONS was smaller than that of HLRTA* and the number of actions to convergence of HLRTA* was smaller than that of LRTA*. In addition, we gained two other important insights:

– The number of trials to convergence of HLRTA* was larger than that of LRTA* but the number of actions in the first trial of HLRTA* was smaller than that of LRTA*. The opposite was true for FALCONS. Thus, the number of trials to convergence is a weakness of HLRTA* and the number of actions in the first trial is a weakness of FALCONS. Figure 1 illustrates this observation for one of the fifteen experimental cases, a four-connected gridworld with the Manhattan distance heuristic. The figure graphs the number of actions for each trial. The graph for HLRTA* started below that of LRTA*,

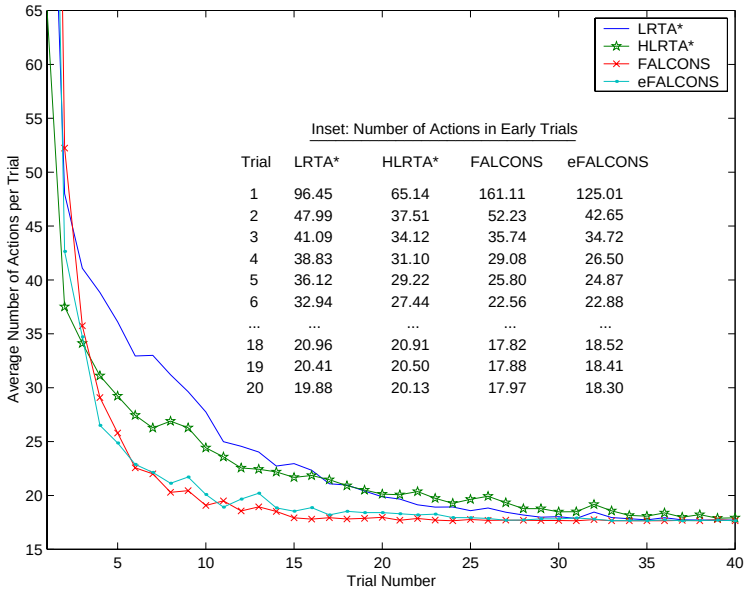


Fig. 1. Comparison of HLRTA* and FALCONS with LRTA* in a Four-Connected Gridworld with Manhattan Distance Heuristic

rose above it after the eighteenth trial (see inset) and then remained above it until the end of learning. The graph for FALCONS, on the other hand, started above that of LRTA*, dropped below it after the second trial and then remained below it until the end of learning.

– The improvement in number of actions until convergence of HLRTA* over LRTA* was smaller in the most informed cases than that of HLRTA* over LRTA* in the least informed cases. The opposite was true for FALCONS. Thus, the number of actions to convergence is a weakness of HLRTA* in the most informed cases and a weakness of FALCONS in the least informed cases.

Thus, there are two reasons for combining HLRTA* and FALCONS. First, the resulting real-time search method could reduce the number of actions to convergence by reducing the number of actions for early trials below that of FALCONS and the number of actions for later trials below that of HLRTA*. Second, the resulting real-time search method could be less sensitive to the level of informedness of the heuristic function and thus perform better across all experimental cases.

3 eFALCONS

LRTA* associates an h-value with every state to estimate the goal distance of the state (similar to the h-values of A*). LRTA* always first updates the h-value of the current state (value-update rule) and then uses the h-values of

Table 2. Comparison of eFALCONS with LRTA*, HLRTA* and FALCONS

Performance Measure	Average Over	Speedup of eFALCONS over LRTA*	HLRTA*	FALCONS
Number of Actions to Convergence	All 15 Cases	21.31%	19.34%	2.18%
	7 Most Informed Cases	28.73%	29.01%	0.00%
	7 Least Informed Cases	13.52%	9.53%	5.16%
Number of Trials to Convergence	All 15 Cases	36.79%	41.44%	-12.11%
	7 Most Informed Cases	38.13%	44.38%	-15.77%
	7 Least Informed Cases	34.29%	37.10%	-10.36%
Number of Actions in First Trial	All 15 Cases	-15.01%	-25.70%	4.72%
	7 Most Informed Cases	-36.86%	-56.21%	7.95%
	7 Least Informed Cases	0.38%	1.28%	0.53%

the successors to move to the successor believed to be on a minimum-cost path from the current state to the goal (action-selection rule). HLRTA* introduces h_s-values in addition to the h-values and uses them to modify the value-update rule of LRTA* so that the h-values converge faster to the goal distances. FALCONS, on the other hand, introduces g- and f-values in addition to the h-values (similar to the g- and f-values of A*) and uses them to modify the action-selection rule of LRTA* so that it moves to the successor believed to be on a shortest path from the start to the goal. eFALCONS, shown in Figure 2, then uses the value-update rule of HLRTA* for both the g- and h-values and the action-selection rule of FALCONS. eFALCONS and FALCONS access only the successors and predecessors of the current state, while LRTA* and HLRTA* access only the successors of the current state. Thus, all four real-time search methods access the same states on undirected graphs. A more detailed description of eFALCONS together with proofs of its properties is given in [4].

4 Empirical Study of eFALCONS

Table 2 compares eFALCONS with LRTA*, HLRTA*, and FALCONS. More detailed results are given in [4], including significance results obtained with the paired-samples Z test at the five-percent confidence level. The table demonstrates two advantages of eFALCONS over HLRTA* and FALCONS:

1. The number of trials to convergence of eFALCONS was 41.44 percent smaller than that of HLRTA*, and the number of actions in the first trial of eFALCONS was 4.72 percent smaller than that of FALCONS. We pointed out earlier that the number of trials to convergence was a weakness of HLRTA* and the number of actions in the first trial was a weakness of FALCONS. Thus, eFALCONS mitigates the weaknesses of both HLRTA* and FALCONS across performance measures. Indeed, Figure 1 shows that the number of actions of eFALCONS was smaller than that of FALCONS in the early trials and smaller than that of HLRTA* in the later trials. As a consequence, the number of actions to convergence of eFALCONS was 19.34 percent smaller than that of HLRTA* and 2.18 percent smaller than that of FALCONS. Thus, combining the value-update rule of HLRTA* and the action-selection

In the following, S denotes the finite state space; $s_{start} \in S$ denotes the start state; and $s_{goal} \in S$ denotes the goal state. $succ(s) \subseteq S$ denotes the set of successors of state s, and $pred(s) \subseteq S$ denotes the set of its predecessors. $c(s, s') > 0$ denotes the cost of moving from state s to successor $s' \in succ(s)$. We use the following conventions: $\arg\min_{s'' \in \emptyset}(\cdot) := \bot$, $\max_{s'' \in \emptyset}(\cdot) := -\infty$, and $\min_{s'' \in \emptyset}(\cdot) := \infty$. We use the following abbreviations, where $\bot$ means "undefined:"

$$\forall s \in S \text{ and } r \in succ(s): \quad h_s(r) := \begin{cases} h(r) & \text{if } dh(r) = \bot \text{ or } dh(r) \neq s \\ sh(r) & \text{otherwise,} \end{cases}$$

$$\forall s \in S \text{ and } r \in pred(s): \quad g_s(r) := \begin{cases} g(r) & \text{if } dg(r) = \bot \text{ or } dg(r) \neq s \\ sg(r) & \text{otherwise, and} \end{cases}$$

$$\forall r \in S: \quad f(r) := \max(g(r) + h(r), h(s_{start})).$$

The values are initialized as follows: $\forall r \in S$: $g(r) := h(s_{start}, r)$ and $h(r) := h(r, s_{goal})$, where $h(r, r')$ is a heuristic estimate of the distance from $r \in S$ to $r' \in S$. Furthermore, $\forall r \in S$: $dg(r) := \bot$, $dh(r) := \bot$, $sg(r) := \bot$, and $sh(r) := \bot$.

1. $s := s_{start}$.
2. $s' := \arg\min_{s'' \in succ(s)} f(s'')$.
   Break ties in favor of a successor s' with the smallest value of $c(s, s') + h_s(s')$.
   Break remaining ties arbitrarily (but systematically).

3 a. $p := \arg\min_{s'' \in pred(s)}(g_s(s'') + c(s'', s))$.
   $n := \arg\min_{s'' \in succ(s)}(c(s, s'') + h_s(s''))$.

   b. Perform the following assignments in parallel:

$g(s) :=$ if $s = s_{start}$ then $g(s)$
else $\max(g(s),$
$\quad g_s(p) + c(p, s),$
$\quad \max_{s'' \in succ(s)}(g(s'') - c(s, s'')))$.

$sg(s) :=$ if $s = s_{start}$ then $g(s)$
else $\max(g(s),$
$\quad \min_{s'' \in pred(s) \setminus \{p\}}(g_s(s'') + c(s'', s)),$
$\quad \max_{s'' \in succ(s)}(g(s'') - c(s, s'')))$.

$dg(s) := p$.

$h(s) :=$ if $s = s_{goal}$ then $h(s)$
else $\max(h(s),$
$\quad c(s, n) + h_s(n),$
$\quad \max_{s'' \in pred(s)}(h(s'') - c(s'', s)))$.

$sh(s) :=$ if $s = s_{goal}$ then $h(s)$
else $\max(h(s),$
$\quad \min_{s'' \in succ(s) \setminus \{n\}}(c(s, s'') + h_s(s'')),$
$\quad \max_{s'' \in pred(s)}(h(s'') - c(s'', s)))$.

$dh(s) := n$.

4. If $s = s_{goal}$, then stop successfully.
5. $s := s'$.
6. Go to 2.

Fig. 2. eFALCONS

rule of FALCONS indeed speeds up their convergence. The number of actions to convergence of eFALCONS, for example, is typically over twenty percent smaller than that of LRTA* and, in some cases, even over fifty percent smaller (not shown in the table).

2. The number of actions to convergence of eFALCONS was 29.01 percent smaller than that of HLRTA* in the most informed cases and 5.16 percent smaller than that of FALCONS in the least informed cases. We pointed out earlier that the number of actions to convergence was a weakness of HLRTA* in the most informed cases and a weakness of FALCONS in the least informed cases. Thus, eFALCONS mitigates the weaknesses of both HLRTA* and FALCONS across the levels of informedness of the heuristic function.

5 Conclusions

In this paper, we presented eFALCONS, a real-time search method that is similar to LRTA* but uses the value-update rule of HLRTA* and the action-selection rule of FALCONS. We showed experimentally that eFALCONS converges to a minimum-cost plan with fewer actions than LRTA*, HLRTA*, and FALCONS. For example, its number of actions to convergence is typically over twenty percent smaller than that of LRTA* and, in some cases, even over fifty percent smaller. It is future work to combine eFALCONS with ε-search to speed up its convergence even more by sacrificing the optimality of the resulting plan.

References

1. S. Edelkamp and J. Eckerle. New strategies in real-time heuristic search. In *Proceedings of the AAAI-97 Workshop on On-Line Search*, pages 30–35. AAAI Press, 1997.
2. D. Furcy and S. Koenig. Speeding up the convergence of real-time search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 891–897, 2000.
3. D. Furcy and S. Koenig. Speeding up the convergence of real-time search: Empirical setup and proofs. Technical Report GIT-COGSCI-2000/01, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), 2000.
4. D. Furcy and S. Koenig. eFALCONS: Speeding up the convergence of real-time search even more. Technical Report GIT-COGSCI-2001/04, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), 2001.
5. T. Ishida. *Real-Time Search for Learning Autonomous Agents*. Kluwer Academic Publishers, 1997.
6. T. Ishida and M. Shimbo. Improving the learning efficiencies of real-time search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 305–310, 1996.
7. R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.
8. P. Thorpe. A hybrid learning real-time search algorithm. Master's thesis, Computer Science Department, University of California at Los Angeles, Los Angeles (California), 1994.

Time-Optimal Planning in Temporal Problems*

Antonio Garrido, Eva Onaindía and Federico Barber

Dpto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
{agarridot,onaindia,fbarber}@dsic.upv.es

Abstract. This paper presents TPSYS, a *Temporal Planning SYStem*, which arises as an attempt to combine the ideas of Graphplan and TGP to solve temporal planning problems more efficiently. TPSYS is based on a three-stage process. The first stage, a preprocessing stage, facilitates the management of constraints on duration of actions. The second stage expands a temporal graph and obtains the set of temporal levels at which propositions and actions appear. The third stage, the plan extraction, obtains the plan of minimal duration by finding a proper flow of actions.

Key words: planning, temporal planning, reasoning about actions

1 Introduction

In real world planning problems which deal with time, it is necessary to discard the assumption that actions have the same duration. For instance, it is clear that in a logistics domain the action `fly plane(London, Moscow)` is longer than `fly plane(London, Paris)`. Hence, dealing with temporal planning problems requires to handle more complex constraints because it is necessary to select the right execution times for actions. Consequently, an important issue in temporal planning is to guarantee the plan which minimizes the global duration.

This paper builds on the work of Smith and Weld (the *Temporal Graphplan* algorithm, TGP, presented in [6]) and examines the general question of including temporality on actions in a Graphplan-based approach [1] by guaranteeing the plan of minimal duration. We present a *Temporal Planning SYStem* (TPSYS) which consists of three stages: a *preprocessing* stage, a temporal graph expansion stage and a plan extraction stage. The main features of TPSYS are:

- It is able to handle overlapping actions of different duration and guarantees the optimal plan, i.e. the plan of minimal duration.
- It defines a new classification of mutual exclusion relations: *static* mutexes which are time independent and *dynamic* mutexes which are time dependent.
- It expands a relaxed temporal graph (from now on *TG*), without maintaining `no − op` actions nor delete-edges, through temporal levels. Then, it performs a plan extraction (from now on *PE*) stage by selecting the appropriate actions in the *TG* to achieve the problem goals.

* This work has been partially supported by the Project n. 20010017 - *Navigation of Autonomous Mobile Robots* of the Universidad Politécnica de Valencia.

2 Related Work

Although temporal features in planning are not usually managed by classical planners, one of the first temporal planners on the last decade was O-Plan [2] which integrates both planning and scheduling processes into a single framework. Other planners, such as IxTeT [4], deal with resource availability and temporal constraints to represent constraints on time points. An attempt to integrate planning and scheduling is performed in HSTS (*Heuristic Scheduling Testbed System* [5]) which defines an integrated framework to solve planning and scheduling tasks. This system uses multi-level heuristic techniques to manage resources under the constraints imposed by the action schedule. The parcPLAN approach [3] manages multiple capacity resources with actions which may overlap, instantiating time points in a similar way to our approach.

TGP [6] introduces a complex mutual exclusion reasoning to handle actions of differing duration in a Graphplan context. TPSYS combines features of both Graphplan and TGP and introduces new aspects to improve performance. The reasoning on *conditional* mutex (involving time mutex) between actions, propositions and between actions and propositions is managed in TGP by means of inequalities which get complex in some problems and may imply an intractable reasoning on large problems [6]. On the contrary, the reasoning process in TPSYS is simplified thanks to the incorporation of several improvements:

- Static mutex relations between actions and between actions and propositions are calculated in a preprocessing stage because they only depend on the definition of the actions.
- TPSYS uses a multi-level temporal planning graph as Graphplan where each level represents an instant of time. While in TGP actions and propositions are only annotated with the first level at which they appear in the planning graph, TPSYS annotates all different instances of actions and propositions produced along time. The compact encoding of TGP reduces vastly the space costs but it increases the complexity of the search process, which may traverse cycles in the planning graph. However, the *PE* in TPSYS is straightforward because it merely consists of obtaining the plan as an acyclic *flow* of actions throughout the *TG*.

3 Our Temporal Planning SYStem

In TPSYS, a temporal planning problem is specified as a 4-tuple $\{\mathcal{I}_s, \mathcal{A}, \mathcal{F}_s, \mathcal{D}_{\max}\}$, where $\mathcal{I}_s$ and $\mathcal{F}_s$ represent the initial and final situation respectively, $\mathcal{A}$ represents the set of actions (with positive duration), and $\mathcal{D}_{\max}$ stands for the maximal duration of the plan required by the user. Time is modelled by $\mathbb{R}^+$ and their chronological order. A temporal proposition is represented by $<\mathtt{p},\mathtt{t}>$ where $\mathtt{p}$ denotes the proposition and $\mathtt{t} \in \mathbb{R}^+$ represents the time at which $\mathtt{p}$ is produced. Hence, $\mathcal{I}_s$ and $\mathcal{F}_s$ are formed by two set of temporal propositions $\{<\mathtt{p_i},\mathtt{t_i}> / \mathtt{t_i} \leq \mathcal{D}_{\max}\}$.

Action	Duration	Precs.	Effects
ld(B1,BC,H)	5	at(B1,H) at(BC,H) free(BC)	in(B1,BC) ¬at(B1,H) ¬free(BC)
mv(BC,H,U)	5	at(BC,H)	at(BC,U) ¬at(BC,H)
uld(B1,BC,U)	2	in(B1,BC) at(BC,U)	at(B1,U) free(BC) ¬in(B1,BC)

Table 1. Simplified *Briefcase* domain: necessary actions to achieve the goal at(B1,U)

We will make use of the action domain defined in Table 1, which presents a description of the actions of the *Briefcase* domain, to show the behaviour of our system. Only three actions are defined, those which are necessary to transport a book (B1) from home (H) to university (U) by using a briefcase (BC).

3.1 First Stage: Preprocessing

TPSYS calculates the static mutual exclusions which will allow us to speed up the following two stages. A mutex relationship between actions is defined as in Graphplan [1]. Mutex between propositions appears as a consequence of mutex between actions. Thus, two propositions p and q are mutex if all actions that achieve p are mutex with all actions that achieve q.

Definition 1. ***Static mutex between actions.*** *Actions* a *and* b *are statically mutex if they cannot be executed in parallel (Graphplan's interference). For instance, in Table 1, actions* $\mathtt{ld}(\mathtt{B1},\mathtt{BC},\mathtt{H})$ *and* $\mathtt{uld}(\mathtt{B1},\mathtt{BC},\mathtt{U})$ *are statically mutex because of the conflicting effect* $\mathtt{in}(\mathtt{B1},\mathtt{BC})$.

Definition 2. ***Static ap-mutex (static action/proposition mutex).*** *One action* a *is statically ap-mutex with a proposition* p *iff* $\mathtt{p} \in \mathtt{del} - \mathtt{effs}(\mathtt{a})$. *For instance,* $\mathtt{ld}(\mathtt{B1},\mathtt{BC},\mathtt{H})$ *is ap-mutex with* $\mathtt{at}(\mathtt{B1},\mathtt{H})$ *and* $\mathtt{free}(\mathtt{BC})$ *in Table 1.*

3.2 Second Stage: Temporal Graph Expansion

Definition 3. ***Temporal graph (TG).*** *A TG is a directed, layered graph with proposition and action nodes, and precondition- and add-edges following the same structure as Graphplan. Each level is labelled with a number representing the instant of time at which propositions are present and actions start their execution. Levels are ordered by their instant of time.*

Definition 4. ***Instance of an action.*** *We define an instance of an action* a *as the triple <*a,s,e*> where* a *denotes the action and* $\mathtt{s},\mathtt{e} \in \mathbb{R}^+$ *represent the time when the instance starts and ends executing, respectively (*$\mathtt{e} = \mathtt{s} + duration(\mathtt{a})$*).*

Definition 5. ***Proposition level.*** *A proposition level $P_{[t]}$ is formed by the set of temporal propositions* $\{<\mathtt{p_i},\mathtt{t_i}>/\mathtt{t_i} \leq \mathtt{t}\}$ *present at time* `t` *which verify* $<\mathtt{p_i},\mathtt{t_i}> \in \mathcal{I}_s \vee \exists <\mathtt{a_i},\mathtt{s_i},\mathtt{e_i}>/\mathtt{p_i} \in \mathtt{add}-\mathtt{effs}(\mathtt{a_i}),\ \mathtt{e_i} = \mathtt{t_i}$.

Definition 6. ***Dynamic mutex between temporal propositions at*** $P_{[t]}$**.** *Let* $\{<\mathtt{a_i},\mathtt{s_i},\mathtt{t_i}>\}$ *and* $\{<\mathtt{b_j},\mathtt{s_j},\mathtt{t_j}>\}$ *be two sets of instances of actions which achieve* $<\mathtt{p},\mathtt{t_i}>,<\mathtt{q},\mathtt{t_j}> \in P_{[t]}$ *respectively. Temporal propositions* $<\mathtt{p},\mathtt{t_i}>$ *and* $<\mathtt{q},\mathtt{t_j}>$ *are dynamically mutex at* $P_{[t]}$ *iff i)* $\forall \alpha, \beta / \alpha \in \{<\mathtt{a_i},\mathtt{s_i},\mathtt{t_i}>\}, \beta \in \{<\mathtt{b_j},\mathtt{s_j},\mathtt{t_j}>\}$, α *and* β *overlap and ii)* $\mathtt{a_i}$ *and* $\mathtt{b_j}$ *are statically mutex. A dynamic mutex expires as new levels are expanded further in the TG.*

Definition 7. ***Action level.*** *An action level $A_{[t]}$ is formed by the set of instances of actions* $\{<\mathtt{a_i},\mathtt{t},\mathtt{e_i}>\}$ *which start their execution at time* `t`.

Proposition 1. *Let $P_{[t]}$ ($t \leq \mathcal{D}_{\max}$) be the earliest proposition level at which all temporal propositions in $\mathcal{F}_s$ are not pairwise dynamically mutex. Under this assumption, no correct plan can be found before time t.*

TPSYS adopts the same conservative model of action as TGP [6]. The second stage expands the *TG* by alternating proposition and action levels through a forward-chaining process. Starting at $P_{[0]}$, the algorithm moves incrementally in time throughout the *TG* generating new action and proposition levels. At each action level $A_{[t]}$, the algorithm generates the entire set of instances of actions which start their execution at t because their preconditions are not dynamically mutex at $P_{[t]}$. After generating each instance of an action, the propositions in $\mathtt{add}-\mathtt{effs}$ are added into the proper proposition level (according to the duration of each action). The *TG* expansion terminates once all temporal propositions in the final situation are present in $P_{[t]}$ and none are pairwise dynamically mutex (i.e. $\mathcal{F}_s$ is satisfied in $P_{[t]}$). If $t > \mathcal{D}_{\max}$ the algorithm outputs '*Failure*' because no feasible plan can be found earlier than $\mathcal{D}_{\max}$.

The resulting *TG* for the domain defined in Table 1 is shown in Fig. 1. Action `uld(B1,BC,U)` cannot start at $A_{[5]}$ because its preconditions `in(B1,BC)` and `at(BC,U)` are dynamically mutex at $P_{[5]}$ and they cannot be simultaneously available until $P_{[10]}$. At $A_{[10]}$, `uld(B1,BC,U)` is applicable thus obtaining the goal `at(B1,U)` at $P_{[12]}$ (terminating the second stage).

3.3 Third Stage: Plan Extraction

This stage is a backward search process throughout the *TG* to extract a feasible plan. Two data structures `PlannedActs` and `GoalsToSatisfy`, which are indexed by a level, are used. `PlannedActs`, which is initialized empty, stores the instances of actions planned at each action level. `GoalsToSatisfy` stores the temporal propositions to be satisfied at each proposition level, and it is initialized by inserting all the temporal propositions in $\mathcal{F}_s$.

Assuming the *PE* process starts from the proposition level $P_{[t]}$ (that is, the search starts from time t in the *TG*), where all temporal goals in $\mathcal{F}_s$ are not dynamically mutex, the algorithm proceeds in the following way:

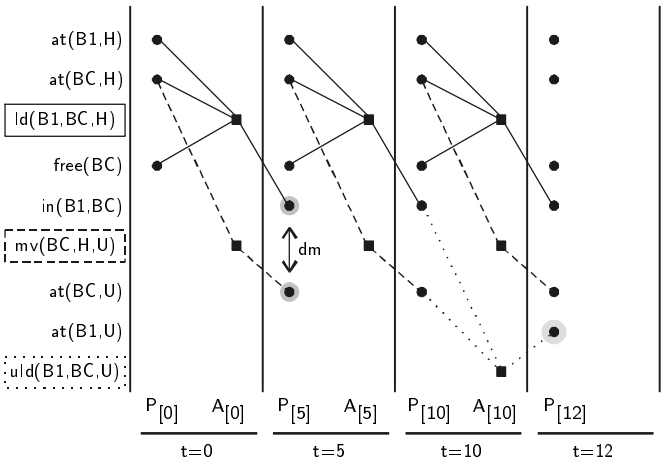


Fig. 1. *Temporal Graph* for the *Briefcase* problem defined in Table 1

1. If $t = 0$ and $\texttt{GoalsToSatisfy}[t] \nsubseteq \mathcal{I}_s$, then fail (backtrack) –this is the base case for the recursive process.
2. If $\texttt{GoalsToSatisfy}[t] = \phi$ then move backwards in time (t =previous level in the TG) and go to step 1 to satisfy the goals at t.
3. Extract a temporal proposition $<\texttt{p},t>$ from $\texttt{GoalsToSatisfy}[t]$.
4. Select an instance of an action $\alpha = <\texttt{a}_\texttt{i},\texttt{s}_\texttt{i},\texttt{e}_\texttt{i}> / \texttt{p} \in \texttt{add} - \texttt{effs}(\texttt{a}_\texttt{i})$, $\texttt{e}_\texttt{i} \leq t$ (*backtracking point* to guarantee completeness). In order to guarantee the correctness of the plan, α is discarded (selecting another instance of an action by *backtracking* to step 4) if at least one of the following conditions holds; i) $\exists\beta = <\texttt{b}_\texttt{j},\texttt{s}_\texttt{j},\texttt{e}_\texttt{j}> \in \texttt{PlannedActs} / \alpha$ and β overlap and $\texttt{a}_\texttt{i}$ and $\texttt{b}_\texttt{j}$ are statically mutex, or ii) $\exists <\texttt{q},\texttt{e}_\texttt{i}> \in \texttt{GoalsToSatisfy} / \texttt{a}_\texttt{i}$ is statically *ap-mutex* with $\texttt{q}$. Otherwise, $\texttt{p}$ is satisfied and the structures $\texttt{PlannedActs}[\texttt{s}_\texttt{i}]$ and $\texttt{GoalsToSatisfy}[\texttt{s}_\texttt{i}]$ are updated with α and $\texttt{precs}(\texttt{a}_\texttt{i})$ respectively. Then, the algorithm goes to step 2 to satisfy another (sub)goal.

Proposition 2. *TPSYS is complete and optimal.*

In TPSYS, all levels at which propositions and actions appear are all generated during the TG expansion. Therefore, if a plan exists for the problem, it will be found in the TG. Additionally, since all instances of actions are considered in the PE process and the TG is expanded through time, the first solution TPSYS finds is the plan of minimal duration.

4 Some Experimental Results

Although comparison between our approach and other planning systems is quite difficult because they are based on different algorithms, we made a comparison

Problem	TPSYS	TGP
tgp-AB-q	4	60
tgp-AB-pq	5	90
tgp-AC-r	4	80
tgp-AC-pr	5	80
tgp-ABDE-r	4	70

Table 2. Results of comparison between TPSYS and TGP (times are in milliseconds)

between TPSYS and TGP on the examples provided by TGP. The experiments (Table 2) were performed in a Celeron 400 MHz with 64 Mb and show the performance of TPSYS is better than TGP for these problems. Consequently, TPSYS seems quite promising to deal with temporal planning problems.

5 Conclusions and Future Work

In this paper we have presented TPSYS, a system for dealing with temporal planning problems. TPSYS contributes on a classification into static and dynamic mutual exclusion relations. This allows to perform a preprocessing stage which calculates static mutexes between actions and between actions and propositions to speed up the following stages. The second stage expands a *TG* with features of both Graphplan and TGP planning graphs. The third stage guarantees that the first found plan has the minimal duration. From our experience and the obtained results we think TPSYS is promising to solve temporal planning problems.

The presented work constitutes a first step towards an integrated system for planning and scheduling. Such a system will be able to manage temporal constraints on actions and to reason on shared resource utilization. Additionally, the system will apply several optimization criteria to obtain the plan of minimal duration or the plan of minimal cost.

References

1. Blum, A.L. and M.L. Furst. "Fast Planning through Planning Graph Analysis," *Artificial Intelligence*, *90*:281–300 (1997).
2. Currie, K. and A. Tate. "O-Plan: the Open Planning Architecture," *Artificial Intelligence*, *52(1)*:49–86 (1991).
3. El-Kholy, A. and B. Richards. "Temporal and Resource Reasoning in Planning: the parcPLAN Approach." *Proc. 12th European Conference on Artificial Intelligence (ECAI-96)*. 614–618. 1996.
4. Ghallab, M. and H. Laruelle. "Representation and Control in IxTeT, a Temporal Planner." *Proc. 2nd Int. Conf. on AI Planning Systems*. 61–67. Hammond, 1994.
5. Muscettola, N. "HSTS: Integrating Planning and Scheduling." *Intelligent Scheduling* edited by M. Zweben and M.S. Fox, 169–212, Morgan Kaufmann, 1994.
6. Smith, D.E and D.S. Weld. "Temporal Planning with Mutual Exclusion Reasoning." *Proc. 16th Int. Joint Conf. on AI (IJCAI-99)*. 326–337. 1999.

Randomization and Restarts in Proof Planning

Andreas Meier[1] Carla P. Gomes[2] Erica Melis[1]

[1] Fachbereich Informatik
Universität des Saarlandes
66041 Saarbrücken, Germany
{ameier|melis}@ags.uni-sb.de

[2] Computer Science Department
Cornell University
Ithaca, NY 14853, USA
gomes@cs.cornell.edu

1 Introduction

Proof planning considers mathematical theorem proving as a planning problem. It has enabled the derivation of mathematical theorems that lay outside the scope of traditional logic-based theorem proving systems. One of its strengths comes from heuristic mathematical knowledge that restricts the search space and thereby facilitates the proving process for problems whose proofs belong in the restricted search space. But this may exclude solutions or restrict the kinds of proofs that can be found for a given problem.

We take a different perspective and investigate problem classes for which little or no heuristic control knowledge is available and test the usage of randomization and restart techniques. Our approach to control in those mathematical domains is based on investigations on so-called *heavy-tailed distributions* ([4, 3, 2]). Because of the non-standard nature of heavy-tailed cost distributions the controlled introduction of randomization into the search procedure and quick restarts of the randomized procedure can eliminate heavy-tailed behavior and can take advantage of short runs. To apply these techniques to the complicated domains of proof planning, the first task was to find problem classes for which proof planning exhibits an unpredictable run time behavior, i.e., with heavy-tailed cost distributions. Secondly, the experiments provided the basis for determining suitable *cutoff values*, i.e., the time interval after which a running proof attempt is interrupted and a new attempt is started. Finally, we designed a new control strategy which dramatically boosts the performance of our proof planner for a class of problems for which proof planning exhibits heavy-tailed cost behavior.

2 Proof Planning

A proof planning problem is defined by an *initial state* specified by the proof assumptions, the *open goal* given by the theorem to be proved, and a set of *operators*[1]. A mathematical proof corresponds to a plan that leads from the initial state to the goal state.

For a very basic example of an operator in proof planning consider the $=Subst$ operator. Its purpose is to replace occurrences of terms with respect to given equations. $=Subst$ is applicable during the planning process if a current goal is a term $t[a]$ that contains an occurrence of a term a and there is an assumption that is an equation with a as one side and another term b as the other side. The application of $=Subst$ reduces then goal $t[a]$ to the new goal $t[b]$ which is the same term as $t[a]$ but the occurrence of a is replaced by an occurrence of b.

The proof planning approach developed in this paper is implemented in the ΩMEGA system [8, 7]. ΩMEGA employs backward chaining as its main planning strategy. That is, the planner continuously tries to reduce open goals by applying an operator that has an appropriate effect, which in turn might result in one or more new open goals and so on. Initially, the only open goal is the theorem. During this planning process there are several choice points such as which goal should be tackled or which operator should be applied in the next step.

3 The Domain of Residue Classes

In this section, we describe the domain of residue classes over the integers. A detailed description of the whole domain can be found in [6].

The Residue Class Domain A residue class set RS_n over the integers is the set of all congruence classes modulo an integer n, i.e., $\mathbb{Z}_n$, or an arbitrary subset of $\mathbb{Z}_n$. Concretely, we can deal with sets of the form $\mathbb{Z}_3, \mathbb{Z}_5, \mathbb{Z}_3 \setminus \{\bar{1}_3\}, \ldots$ where $\bar{1}_3$ denotes the congruence class 1 modulo 3. Binary operations $\circ$ on a residue class set are either $\bar{+}, \bar{-}, \bar{*}$ which are the addition, subtraction, and multiplication on residue classes or functions composed from these connectives, e.g. $(\bar{x} \bar{*} \bar{y}) \bar{+} (\bar{y} \bar{+} \bar{x})$. For given residue class set and binary operation we can examine their basic algebraic properties (is the set RS_n closed with respect to the binary operation $\circ$, is it associative, does it have a unit element etc.) and classify them in terms of groups, monoids, etc. Moreover, we are interested in classifying structures into equivalence classes of isomorphic structures. During this classification process we have to prove proof obligations stating that two structures $(RS^1_{n_1}, \circ_1)$ and $(RS^2_{n_2}, \circ_2)$ are isomorphic or not. Thereby, two structures $(RS^1_{n_1}, \circ_1)$ and $(RS^2_{n_2}, \circ_2)$ are isomorphic if there exists a total function $h : RS_{n_1} \rightarrow RS_{n_2}$ such that h is injective, surjective, and is a homomorphism with respect to $\circ_1$ and $\circ_2$. A function h is a homomorphism, if $h(x \circ_1 y) = h(x) \circ_2 h(y)$ holds for all $x, y \in RS_{n_1}$. A *non-isomorphism problem* is formalized as $\neg iso(RS^1_{n_1}, \circ_1, RS^2_{n_2}, \circ_2)$, where iso abbreviates $isomorphic$.

Two Proof Strategies We developed several proof techniques to tackle these non-isomorphism problems in ΩMEGA. We will focus here on two of those techniques (1) proof by case analysis and (2) proof by contradiction.

(1) The case analysis strategy is a basic but reliable approach to prove a property of a residue class structure. Its essence is a proof by cases. It exhaustively checks all instances of a conjecture. Since residue class sets are finite, only finitely many instances have to be considered. For non-isomorphism problems the top-most case split is to check for each possible function from the one residue class set into the other one that it is either not injective, not surjective, or not a homomorphism.

(2) An alternative proof strategy creates a proof by contradiction. It assumes that there exists a function $h{:}RS^1_{n_1} \rightarrow RS^2_{n_2}$ which is an isomorphism and thus, in particular, an injective homomorphism. It derives the contradiction by proving that there are two elements $c_1, c_2 \in RS^1_{n_1}$ with $c_1 \neq c_2$ but $h(c_1) = h(c_2)$ which contradicts the assumption of injectivity of h. Note, that the proof is with respect to all possible homomorphisms h and we do not have to give a particular mapping. In the remainder of

the paper we call the described proof technique to tackle non-isomorphism proofs the `NotInjNotIso` technique.

We briefly explain the `NotInjNotIso` strategy for the example that $(\mathbb{Z}_5, \bar{x}\bar{*}\bar{y})$ is not isomorphic to $(\mathbb{Z}_5, \bar{x}\bar{+}\bar{y})$. The strategy first constructs the situation for the indirect argument. From the hypothesis that the two structures are isomorphic follow the two assumptions that there exists a function h that is injective and a homomorphism. By the first assumption a contradiction can be concluded when we are able to show that h is not injective.

The planner continues by applying a method to the second assumption, that introduces the homomorphism equation $h(x\bar{*}y) = h(x)\bar{+}h(y)$ instantiated for every element of the domain as new assumptions. In the above example 25 equations like

$h(\bar{0}_5) = h(\bar{0}_5)\bar{+}h(\bar{1}_5)$ for $x = \bar{0}_5, y = \bar{1}_5$ (a)

$h(\bar{0}_5) = h(\bar{0}_5)\bar{+}h(\bar{0}_5)$ for $x = \bar{0}_5, y = \bar{0}_5$ (b)

are introduced. From this set of instantiated homomorphism equations the `NotInjNotIso` strategy tries to derive that h is not injective. To prove this, it has to find two witnesses c_1 and c_2 such that $c_1 \neq c_2$ and $h(c_1) = h(c_2)$. In our example $\bar{0}_5$ and $\bar{1}_5$ are chosen for c_1 and c_2, respectively, which leads to $h(\bar{0}_5) = h(\bar{1}_5)$. This goal is transformed into the equation $h(\bar{0}_5)\bar{+}h(\bar{0}_5)\bar{+}h(\bar{0}_5)\bar{+}h(\bar{0}_5)\bar{+}h(\bar{0}_5)\bar{+}h(\bar{1}_5) = h(\bar{1}_5)$ by successively applying equations from the equation system with the operator $=Subst$. First, equation (a) is applied to the left hand side of the equation which results in $h(\bar{0}_5)\bar{+}h(\bar{1}_5) = h(\bar{1}_5)$. Then equation (b) is applied four times to occurrences of $h(\bar{0}_5)$ on the left hand side. The final goal is closed by an application of the operator $Solve{-}Equation$ which calls the Computer Algebra System MAPLE to evaluate the equation. The final equation holds since $5\bar{*}h(\bar{0}_5)$ equals $\bar{0}_5$ modulo 5. The choice of the next instantiated homomorphism equation to be applied is guided by a heuristic described in [5].

4 Experimental Results

The experiments were conducted with 160 non-isomorphism problems for the residue class set $\mathbb{Z}_5$. We decided for the residue class set $\mathbb{Z}_5$ because its cardinality is small enough to obtain solution statistics in a reasonable time. Problems from this class are:
1. $\neg iso(\mathbb{Z}_5, x\bar{*}y, \mathbb{Z}_5, x\bar{+}y)$,
2. $\neg iso(\mathbb{Z}_5, x\bar{-}y, \mathbb{Z}_5, (x\bar{-}y)\bar{+}(x\bar{-}y))$.
The overall experimental effort was around one month of cpu time on a 32 node compute cluster. A detailed description of all experiments can be found in [5].

4.1 Randomization and Heavy-Tailed Behavior

First let us consider the `NotInjNotIso` strategy because this strategy leads to the most interesting proof planning behavior in the residue class domain.The application of the `NotInjNotIso` strategy to all problems of the testbed solved 108 of the 160 instances (67.5%) (2 hour time limit per proof attempt). The runs revealed a surprisingly high variance in the performance of this strategy on the different problems of the testbed. On some of the problems it succeeded very fast and produced short proof plans consisting only of a few applications of $=Subst$, whereas on other problems the planning process took much longer and resulted in proof plans with many applications of $=Subst$. Furthermore, for over 30% of the instances no proof was found in 2 hours.

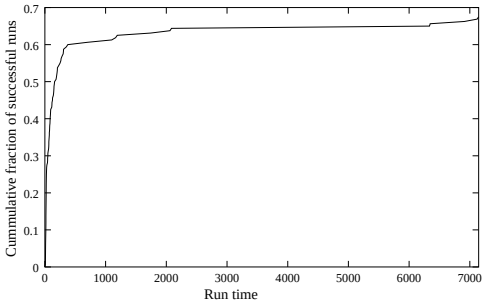


Fig. 1. Run time distribution over testbed without randomization.

Table 1 displays the performance extrema for this deterministic proof by contradiction strategy on the testbed as well as the mean values over all successful runs. The values in brackets give the deviation from the mean. Fig. 1 shows the underlying distribution of the run time for these experiments. In fact, the distribution exhibits *heavy-tailed* behavior [2] which is manifested in the long tail of the distribution stretching for several orders of magnitude. Gomes *et al.* have shown that one can take advantage of the large variations in run time of such heavy-tailed distributions by introducing an element of randomness into the search process, combined with a restart strategy.

Costs	Mean	Min.	Max.
Proof length	55	45 (18.2%)	83 (50.9%)
Run Time	483	$8(98\%)$	$7145(1380\%)$

Table 1. Statistics for successful runs (108 out of 160) on testbed using deterministic strategy.

A key criterion for the success of such a randomization and restart approach is a large variance in different randomized runs with the same instance. To explore this issue, we considered multiple runs on a single instance by introducing a stochastic element into the planning process. Typically, the heuristic for choosing the next instantiated homomorphism equation to be applied ranks several equations equally good. When faced with such equally ranked equations, the planner applies them in a random order. This randomized version of the `NotInjNotIso` technique was run 225 times for the problem instance of the testbed:

$$\neg iso(\mathbb{Z}_5, (\bar{x} \bar{+} \bar{y}) \bar{+} \bar{2}_5, \mathbb{Z}_5, (\bar{2}_5 \bar{*} (\bar{x} \bar{+} \bar{y})) \bar{+} \bar{2}_5)$$

(in the remainder of this section we refer to this problem as the *standard problem*). Interestingly, the run time distribution of the randomized proof search by contradiction on the single instance also exhibits heavy-tailed behavior similar to Fig. 1 (see [5] for a detailed analysis). This indicates an inherent variance in the search process of the strategy.

Given this result, we can now use a restart strategy to improve the proof search performance. Fig. 1 shows that the ascend of the cumulative cost distribution function is very steep at the beginning but becomes very flat beyond approximately 300 seconds. This steep ascend at the beginning indicates that there is a large fraction of short and successful runs whereas the flat ascend after 300 seconds provides evidence that the probability of finding a proof plan decreases considerably. Hence, it is advantageous

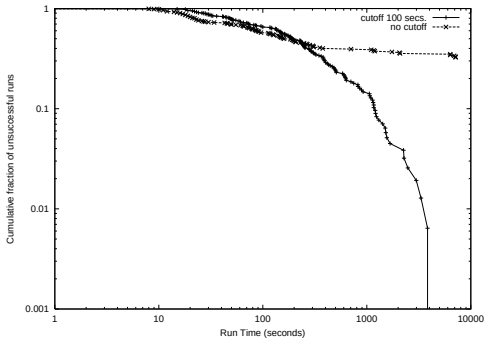


Fig. 2. Log-Log plots of run time distribution over testbed with and without randomization.

to perform a sequence of restarts on a single instance (with a predefined cutoff) until reaching a successful run or the total time limit, instead of performing a single long run.

Based on an analysis of the underlying distributions of the experiments for the full testbed and for the standard problem we considered several cutoff values, using a binary search strategy. The cutoff value of 100 provided the best results. The planner found proof plans for 156 of the 160 problems (97.5%) in an average time of 473.4 seconds. For the four remaining unsolved problems MAPLE does not provide any substitution hint and, thus, the proof by contradiction strategy becomes quite ineffective.

Fig. 2 plots the run time distribution of the resulting restart strategy with cutoff 100 (log-log scale) on the problems of the testbed. The restart data is given by the curve that drops rapidly. The figure also shows the run time distribution of the deterministic strategy. The sharp drop of the run time distribution of the restart strategy clearly indicates that this strategy does not exhibit heavy tailed behavior.

In previous applications of randomization and restarts in combinatorial domains run time has been the key issue [2]. In the case of proof planning, an additional important issue is the length of the proof discovered by the system: shorter proofs are generally more elegant than long proofs. An interesting aspect of the application of randomization and restart strategies that is novel in our context is the fact that it leads to a variety of proof lengths for the same problem instance. For instance, for our standard problem instance, we found a range of proofs from proofs consisting of 47 to 78 nodes. Such a degree of variance is unusual for proofs generated by proof planning.

Having a set of proof planning operators and a flexible control that includes randomization the planner can generate a variety of proofs. This greatly enhances the ability of the system to find proofs and increase the overall robustness of the theorem proving system.

4.2 Case Analysis Strategy

This strategy explores all possible mappings between the structures. Since the goal is to prove a non-isomorphism, the prover needs to establish that no mapping is an isomorphism. Obviously, this strategy is computationally very expensive and, as our experiments show, it is practically infeasible for structures of cardinality larger than four. There still is the question as to whether randomization may be of use in this context.

Table 2 shows that there is still some variation in run time and proof length (100 randomized runs on a single problem instance from $\mathbb{Z}_3$) due to different search pruning effects, but the variations are small compared to those encountered for the proof by contradiction strategy. Further analysis (see [5]) shows that the underlying distribution is not heavy-tailed and therefore a restart strategy would not boost the performance significantly.

Costs	Mean	Min.	Max.
Proof Length	598	540 (9.7%)	684 (14.4%)
Run Time	2456	1110 (54.8%)	4442 (80.9%)

Table 2. Randomized version of the case analysis strategy.

5 Conclusions

The analysis of the cost distributions of proof planning attempts for a class of theorems and on the detection of *heavy-tailed* behavior gave rise to an application of randomization and restarts techniques. The experimental part of the investigations includes a study of two different planning strategies and the determination of cut-off values for the restart. As a conclusion, we have introduced new kind of control knowledge into the proof planning process, a much larger fraction of problem instances became solvable (from 67.5% to 97.5%), and a variety of proofs can be generated for a problem. The application of randomization and restart techniques makes the search process more robust even when the size of the search spaces involved grows super-exponentially. We described in this paper experiments with non-isomorphism problems of the residue class set $\mathbb{Z}_5$. We obtained analogous results on non-isomorphism problems of the residue class sets $\mathbb{Z}_2$, $\mathbb{Z}_3$, $\mathbb{Z}_4$ and $\mathbb{Z}_6$ (see [5]).

Proof planning can benefit from these investigations in general because they provide a stochastic approach to semi-automatically designing control knowledge and because this kind of control knowledge can augment the mathematically motivated control knowledge previously used in proof planning.

References

1. A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In E. Lusk and R. Overbeek, editors, PROCEEDINGS *of CADE–9*, volume 310 of *LNCS*. Springer, Germany, 1988.
2. C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasonong*, 24:67–100, 2000.
3. C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of AAAI-98*, pages 431–437, 1998.
4. C. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of AIPS'98*, pages 208–213, 1998.
5. A. Meier. Randomization and heavy-tailed behavior in proof planning. SEKI-Report SR-00-03 (SFB), Universität des Saarlandes, Saarbrücken, Germany, 2000.
6. A. Meier, M. Pollet, and V. Sorge. Exploring the domain of residue classes. SEKI-Report SR-00-04 (SFB), Universität des Saarlandes, Saarbrücken, Germany, 2000.
7. E. Melis and A. Meier. Proof planning with multiple strategies. In *Proc. of the First International Conference on Computational Logic (CL2000)*, pages 644–659, 2000.
8. E. Melis and J. Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 1999.

Modeling Clairvoyance and Constraints in Real-Time Scheduling

K. Subramani

Department of Computer Science and Electrical Engineering,
West Virginia University,
Morgantown, WV, USA
{ksmani@csee.wvu.edu}

Abstract. Scheduling in Real-time systems differs from scheduling in conventional models in two principal ways: (a) Parameter variability, (b) Existence of complex constraints between jobs. Our work focusses on variable execution times. Whereas traditional models assume fixed values for job execution time, we model execution times of jobs through convex sets. The second feature unique to real-time systems, is the presence of temporal relationships that constrain job execution. Consider for instance the requirement that job 1 should conclude 10 units before job 2. This can be modeled through a simple, linear relationship, between the start and execution times of jobs 1 and 2. In real-time scheduling, it is important to guarantee a priori, the scheduling feasibility of the system. Depending upon the nature of the application involved, there are different schedulability specifications viz. Static, Co-Static and Parametric. Each specification comes with its own set of flexibility issues. In this paper, we present a framework that enables the specification of real-time scheduling problems and discuss the relationship between flexibility and complexity in the proposed model. We motivate each aspect of our model through examples from real-world applications.

1 Introduction

In this paper, we describe the features of our real-time scheduling framework called the `E-T-C` (Execution-Time-Constraints) Real-Time Scheduling model. Real-time scheduling differs from traditional scheduling in two fundamental ways, viz. non-constant execution times and the existence of complex constraints (such as relative timing constraints) between the constituent jobs of the underlying system. A traditional scheduling model such as the one discussed in [14] and [3] assumes that the execution time of a job is a fixed constant. This assumption is not borne out in practice; for instance the running time of an input dependent loop structure such as **for(** $i = 1$ **to** N **)** will depend upon the value of N. Secondly, jobs in a real-time system are often constrained by complex relationships such as: Start job J_a within 5 units of Job J_b completing. Traditional scheduling literature does not accommodate constraints more complex than those that can be represented by precedence graphs.

Our scheduling model is composed of 3 sub-models, viz. the Job model, the Constraint model and the Query model. The Job model describes the type of jobs that we are interested in scheduling. The Constraint model is concerned with the nature of relationships constraining the execution of the jobs. The Query model specifies what it means

for a set of jobs to be schedulable, subject to constraints imposed as per the Constraint model. *An instance of a problem in the* E-T-C *model is specified by instantiating the variables in the sub-models.*

We focus on the following issues:

(a) Designing a framework that enables specification of real-time scheduling problems, and
(b) Studying instantiations of interest in this framework.

The rest of this paper is organized as follows: Section §2 describes the Job model within the E-T-C scheduling framework. The Constraint model is discussed in the succeeding section viz. Section §3. Section §4 details the Query model and presents the 3 types of queries that we consider in this thesis. Each aspect of the E-T-C scheduling framework is motivated through an example from real-time design. A classification scheme for Scheduling problems in the E-T-C model is introduced in §5.

2 Job Model in E-T-C

Assume an infinitely extending time axis, starting at time $t = 0$. This axis is divided into intervals of length L; these intervals are ordered and each interval is called a scheduling window e.g. $[0, L]$ represents the first scheduling window, $[L, 2.L]$ represents the second scheduling window and in general, $[(i-1).L, i.L]$ represents the i^{th} scheduling window. We are given a set of *ordered, non-preemptive,* jobs $\mathcal{J} = \{J_1, J_2, \ldots J_n\}$, with start times $\{s_1, s_2, \ldots, s_n\}$ and execution times $\{e_1, e_2, \ldots, e_n\}$. L is the period of the job-set and all jobs execute periodically in each scheduling window. We remark that non-preemptive jobs form the bulk of real-time applications in mission-critical tasks [11].

3 Constraint Model in E-T-C

The executions of the jobs in the job-set $\mathcal{J}$ (discussed in the above section) are constrained through relationships that exist between their start times and execution times. In the E-T-C model, we permit only linear relationships; thus the constraint system on the job-set is expressed in matrix form as :

$$\mathbf{A}.[\vec{\mathbf{s}}, \vec{\mathbf{e}}] \leq \vec{\mathbf{b}}, \tag{1}$$

where,

- $\vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n]$ is an $n-$vector, representing the start times of the jobs;
- $\vec{\mathbf{e}} = [e_1, e_2, \ldots, e_n]$ is an $n-$vector representing the execution times of the jobs;
- $\mathbf{A}$ is a $m \times 2.n$ matrix of rational numbers, called the *constraint matrix*;
- $\vec{\mathbf{b}} = [b_1, b_2, \ldots, b_m]$ is an $m-$vector of rational numbers,

Observe that System (1) can be rewritten in the form:

$$\mathbf{G}.\vec{\mathbf{s}} + \mathbf{H}.\vec{\mathbf{e}} \leq \vec{\mathbf{b}}, \tag{2}$$

where,

$$\mathbf{G}.\vec{\mathbf{s}} + \mathbf{H}.\vec{\mathbf{e}} = \mathbf{A}.[\vec{\mathbf{s}}, \vec{\mathbf{e}}]$$

We can also use the finish times f_i of jobs in relationships. Since the jobs are non-preemptive, the relation: $s_i + e_i = f_i$ holds for all jobs J_i and hence our expressiveness is not enhanced by the inclusion.

System (1) is a convex polyhedron in the $2.n$ dimensional space, spanned by the start time axes $\{s_1, s_2, \ldots, s_n\}$ and the execution time axes $\{e_1, e_2, \ldots, e_n\}$.

The execution times are independent of the start times of the jobs; however they may have complex interdependencies among themselves. This interdependence is expressed by setting

$$\vec{\mathbf{e}} \in \mathbf{E} \tag{3}$$

where $\mathbf{E}$ is an arbitrary convex set. We regard the execution times as $n-$vectors belonging to the set $\mathbf{E}$.

The ordering on the jobs is obtained by imposing the constraints:

$$s_i + e_i \leq s_{i+1}, \forall i = 1, \ldots, n-1.$$

The ordering constraints are included in the $\mathbf{A}$ matrix in (1).

The Constraint model can be adapted to special situations by restricting either $\mathbf{E}$ or $\mathbf{A}$ or both. The following advantages result from such restrictions:

- A model that more accurately describes the requirements of the current situation,
- Faster algorithms for schedulability queries, and
- More efficient dispatching schemes.

In §3.1, §3.2 and §3.3 we discuss restrictions to the convex set $\mathbf{E}$, while §3.4, §3.5 and §3.6 deal with restrictions to the constraint matrix $\mathbf{A}$.

3.1 The Axis-parallel Hyper-rectangle domain

As specified above, the set $\mathbf{E}$ in the Constraint model can be an arbitrary convex domain. One domain that finds wide applicability is the axis-parallel hyper-rectangle domain (henceforth abbreviated as `aph`). The Maruti Operating System [8–10] estimates running times of jobs by performing repeated *runs* so as to determine upper and lower bounds on their execution time. Accordingly, the running time of job J_i, viz. e_i, belongs to the interval $[l_i, u_i]$, where l_i and u_i denote the lower and upper bounds on the execution time as determined by empirical observation. `These independent range variations are the only constraints on the execution times.` Observe that during actual execution, e_i can take any value in the range $[l_i, u_i]$.

The `aph` domain possesses two useful features:

- A specification that is tractable for this domain is also tractable for arbitrary convex domains [23],
- A specification that is provably "hard" for arbitrary convex domains is also "hard" for this domain [18].

Thus when proving complexity results (especially *hardness results*), it suffices to focus on the `aph` domain only.

3.2 The Polyhedral domain

A feature of machining systems such as the ones discussed in [25] and [7] is the active interdependence of execution times on each other. For instance, the requirement that the sum of the speeds of two axes J_1 and J_2 not exceed k is captured by: $e_1 + e_2 \leq a$. Polyhedral domains are generalizations of the aph domains discussed above.

3.3 Arbitrary Convex Sets

Even polyhedral domains cannot capture the requirements of Power Systems in which there exists quadratic constraints on the execution times. For instance, the spherical constraint $e_1^2 + e_2^2 + \ldots e_n^2 \leq r, r \geq 0$ captures the requirement that the total power spent in the system is bounded by r [2] .

3.4 Standard Constraints

The class of "standard constraints" was introduced in [16], as a restriction to the constraint matrix $\mathbf{A}$ for which the Parametric Schedulability query (see Section §4.3) could be decided efficiently.

Definition 1. *A constraint is said to be a standard constraint, if it can be expressed as a strict difference relationship between at most two jobs. The relationship could be expressed between their start or finish times.*

These constraints are also known as *monotone constraints* in the literature [6]. Standard constraints serve to model relative positioning requirements between two jobs and absolute constraints on a single job. When the constraints are standard, the matrix $\mathbf{G}$ in System (2) is network unimodular [5, 13] and hence the constraint system can be represented as a network graph [21, 4].

The advantage of the network representation is that certain feasibility queries in the primal system can be expressed as shortest-path queries in the corresponding dual network [4]. Standard constraints are widely used to model temporal relationships in flight-control systems [11, 12].

3.5 Network Constraints

Network constraints are a straightforward generalization of standard constraints.

Definition 2. *A constraint is said to be a network constraint, if it can be put in the following form:*

$$a.s_i + b.s_j \leq c.e_i + d.e_j + k, \tag{4}$$

where $a, b, c, d, k \in \Re$.

Network constraints can also be represented as graphs [6, 1]; however the relationships between adjacent vertices form a polyhedron and are not adequately represented through edges, as in the case of standard constraints. Once again, the advantage of the graph representation is the existence of faster algorithms for feasibility checking as opposed to general constraints. Network constraints find wide applicability in approximating certain measures [19].

3.6 Arbitrary Constraints

Job completion statistics such as *Sum of Completion times* and *Weighted Sum of Completion times* of jobs are of interest to the designers of real-time systems [25]. These statistics are aggregate constraints $\sum_{i=1}^{n}(s_i + e_i)$ and cannot be captured through either standard or network constraints.

4 Query model in E-T-C

Goal: *We wish to determine a start time vector $\vec{\mathbf{s}}$, in each scheduling window, such that the constraint system (1) holds (is not violated)* at run-time *for any execution time vector $\vec{\mathbf{e}} \in \mathbf{E}$.*

The above specification (called the *schedulability specification*) is rather vague and is intended to be so; in this section, we shall present three different formalizations of the informal specification above. Each formalization (specification) has a different notion of what it means for a job-set to be schedulable and is characterized by a distinct set of complexity issues and flexibility concerns. *However, in all the specifications the guarantees provided are absolute i.e. if the schedulability query is decided affirmatively, then the constraint set will not be violated at run time.* We also use the terms *schedulability query* and *schedulability predicate* to refer to the schedulability specification.

4.1 Static Scheduling

Static scheduling (also called *Scheduling with no Clairvoyance*) is concerned with deciding the following predicate:

$$\mathcal{P}_s \equiv \exists \vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n] \forall \vec{\mathbf{e}} = [e_1, e_2, \ldots, e_n] \in \mathbf{E} \quad \mathbf{A}.[\vec{\mathbf{s}}, \vec{\mathbf{e}}] \leq \vec{\mathbf{b}} \quad ? \tag{5}$$

In other words, the goal is to determine the existence of a single start-time vector $\vec{\mathbf{s}} \in \Re^n$, such that the constraint system represented by (1) holds. The only information that is available prior to the dispatching of jobs in the i^{th} scheduling window is the knowledge of the execution time domain $\mathbf{E}$.

In [23], we showed that the above proposition can be decided efficiently for arbitrary convex domains. From a computational perspective, query (5) is the easiest to answer. Static Scheduling is the only mode of scheduling at one's disposal, if the dispatcher does not have the power to perform online computations; in fact $O(1)$ dispatching time is one of the advantages of static scheduling [24].

4.2 Co-Static Scheduling

Static scheduling is unduly restrictive in that even simple constraint sets will fail to have static schedules [22]. The restrictiveness of Static Scheduling stems from the insistence on rational solution vectors. If however, the solution vector is allowed to be a function of the execution time vector, then a greater amount of flexibility results. In Co-Static Scheduling (also called *Scheduling with total Clairvoyance*), the assumption is that the execution time vector is known at the start of the scheduling window, although it

may be different in different windows. Accordingly, we wish to decide the following predicate:

$$\mathcal{P}_c \equiv \forall \vec{\mathbf{e}} = [e_1, e_2, \ldots, e_n] \in \mathbf{E} \quad \exists \vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n] \quad \mathbf{A}.[\vec{\mathbf{s}}, \vec{\mathbf{e}}] \leq \vec{\mathbf{b}} \quad ? \tag{6}$$

Co-static scheduling permits maximum flexibility during the dispatching phase, in that if a constraint system is not co-statically schedulable, then it is not schedulable. However, query (6) is `coNP-complete` for arbitrary constraint sets, as shown in [22]. We have recently shown that the co-static schedulable query is solvable in polynomial time for standard and network constraints [20]. Co-static scheduling queries are applicable in Flow-shops [15].

4.3 Parametric Scheduling

Co-static scheduling requires knowledge of the execution time vector for a particular scheduling window, prior to determining the start time vector for that window. This may not be feasible in all real-time systems. Parametric scheduling (also called *Scheduling with limited Clairvoyance*) attempts to provide a balance between the Static an Co-Static scheduling modes. In a parametric schedule, the start time of a job is permitted to depend upon the start and execution times of jobs that have been sequenced before it *and only on those times.* In this mode, we restrict our discussion to `aph` domains, inasmuch as this simple domain preserves the hardness of schedulability queries. Thus, the parametric schedulability predicate is:

$$\mathcal{P}_p \equiv \exists s_1 \; \forall e_1 \; \in [l_1, u_1] \; \exists s_2 \; \forall e_2 \; \in [l_2, u_2] \; \ldots \exists s_n \; \forall e_n \; \in [l_n, u_n] \; \mathbf{A}.[\vec{\mathbf{s}}, \vec{\mathbf{e}}] \leq \vec{\mathbf{b}} \quad ? \tag{7}$$

5 A Taxonomy of Scheduling problems

From the discussion in the above sections, it is clear that in order to specify an instance of a scheduling problem in the `E-T-C` scheduling framework, it is necessary to specify:

- The nature of the execution time domain ($\mathbf{E}$),
- The type of constraints on the jobs ($\mathbf{A}$), and
- A description of the schedulability query ($\mathcal{P}_s, \mathcal{P}_c, \mathcal{P}_p$).

Thus, a problem instance can be specified by instantiating the tuples in the $< \alpha|\beta|\gamma >$ triplet, where,

- α represents the execution time domain $\mathbf{E}$ - The following values are permissible for α:
  - `aph` - $\mathbf{E}$ is an axis-parallel hyper-rectangle,
  - `poly` - $\mathbf{E}$ is a polyhedron
  - `arb` - $\mathbf{E}$ is an arbitrary convex domain.

  Clearly `aph` is the weakest domain in terms of what can be specified and `arb` is the strongest.
- β represents the constraint matrix $\mathbf{A}(\mathbf{G}, \mathbf{H})$ - β can assume the following values:

- `stan` - The constraints are *standard* which implies that $\mathbf{G}$ and $\mathbf{H}$ are network, unimodular matrices.
- `net` - The constraints are *network* which implies that $\mathbf{G}$ and $\mathbf{H}$ have at most two non-zero entries in any row
- `arb` - $\mathbf{G}$ and $\mathbf{H}$ are an arbitrary $m \times n$ rational matrices

Once again `stan` is the weakest constraint class, in terms of real-time constraints that it can model, whereas `arb` is the strongest.

– γ represents the schedulability predicate - The schedulability predicate specifies what it means for a set of jobs to be schedulable; the following values are permitted:
  - `stat` - The query is concerned with static schedulability,
  - `co-stat` - The query is concerned with co-static schedulability,
  - `param` - The query is concerned with parametric schedulability.

Clearly `co-stat` is the most flexible query and `stat` is the least flexible.

Accordingly, $< aph|arb|stat >$ represents an instance of a real-time scheduling problem, in which the execution time domain is an axis-parallel hyper-rectangle, the constraints are arbitrary and the schedulability predicate is static. Our notation scheme is similar to the $< \alpha|\beta|\gamma >$ scheme for traditional scheduling models [14, 3].

6 Offline Analysis versus Online Dispatching

Scheduling algorithms in the `E-T-C` model possess an offline schedulability analyzer and an online dispatching component The analyzer examines the constraints on the system and the type of schedulability query involved, to determine whether a feasible schedule is possible. *This analysis is always carried out offline.* The dispatching component is concerned with determining the exact start times of the jobs in the current scheduling window. *Dispatching is always carried out online.*

For a given instance of a scheduling problem, the offline analyzer is executed exactly once. If the schedulability query is decided affirmatively, the online dispatcher is executed in every scheduling window.

References

1. Bengt Aspvall and Yossi Shiloach. A fast algorithm for solving systems of linear equations with two variables per equation. *Linear Algebra and its Applications*, 34:117–124, 1980.
2. M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. John Wiley, New York, second edition, 1993.
3. P. Brucker. *Scheduling*. Akademische Verlagsgesellschaft, Wiesbaden, 1981.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.
5. G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.
6. Dorit S. Hochbaum and Joseph (Seffi) Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, December 1994.
7. Y. Koren. *Computer Control of Manufacturing Systems*. McGraw-Hill, New York, 1983.

8. S. T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala. The `Maruti` Hard Real-Time Operating System. *ACM Special Interest Group on Operating Systems*, 23(3):90–106, July 1989.
9. D. Mosse, Ashok K. Agrawala, and Satish K. Tripathi. Maruti a hard real-time operating system. In *Second IEEE Workshop on Experimental Distributed Systems*, pages 29–34. IEEE, 1990.
10. D. Mosse, Keng-Tai Ko, Ashok K. Agrawala, and Satish K. Tripathi. Maruti: An Environment for Hard Real-Time Applications. In Ashok K. Agrawala, Karen D. Gordon, and Phillip Hwang, editors, *Maruti OS*, pages 75–85. IOS Press, 1992.
11. N. Muscettola, B. Smith, S. Chien, C. Fry, G. Rabideau, K. Rajan, and D. Yan. In-board planning for autonomous spacecraft. In *The Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS)*, July 1997.
12. Nicola Muscettola, Paul Morris, Barney Pell, and Ben Smith. Issues in temporal reasoning for autonomous control systems. In *The Second International Conference on Autonomous Agents*, Minneapolis, MI, 1998.
13. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.
14. M. Pinedo. *Scheduling: theory, algorithms, and systems*. Prentice-Hall, Englewood Cliffs, 1995.
15. M. Pinedo. *Scheduling: theory, algorithms, and systems*, chapter 5. In [14], 1995.
16. Manas Saksena. *Parametric Scheduling in Hard Real-Time Systems*. PhD thesis, University of Maryland, College Park, June 1994.
17. K. Subramani. *Duality in the Parametric Polytope and its Applications to a Scheduling Problem*. PhD thesis, University of Maryland, College Park, July 2000.
18. K. Subramani. On the complexity of co-static scheduling. Technical Report 2000-0006, West Virginia University, December 2000.
19. K. Subramani. Parametric scheduling for network constraints. In *The Seventh Annual International Computing and Combinatorics Conference*, 2001.
20. K. Subramani. Polynomial time algorithms for co-static scheduling. Technical report, West Virginia University, April 2001. Manuscript in Preparation.
21. K. Subramani and A. K. Agrawala. A dual interpretation of standard constraints in parametric scheduling. In *The Sixth International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, September 2000.
22. K. Subramani and A. K. Agrawala. The parametric polytope and its applications to a scheduling problem. Technical Report CS-TR-4116, University of Maryland, College Park, Department of Computer Science, March 2000.
23. K. Subramani and A. K. Agrawala. The static polytope and its applications to a scheduling problem. 3^{rd} *IEEE Workshop on Factory Communications*, September 2000.
24. I. Tsamardinos, N. Muscettola, and P. Morris. Fast transformation of temporal plans for efficient execution. In *The Fifteenth National Conference on Artificial Intelligence (AAAI-98)*.
25. Y.Koren. Cross-coupled biaxial computer control for manufacturing systems. *ASME Journal of Dynamic Systems, Measurement and Control*, 102:265–272, 1980.

Flexible Dispatch of Disjunctive Plans

Ioannis Tsamardinos[1], Martha E. Pollack[2], and Philip Ganchev[1]

[1] Intelligent Systems Program, University of Pittsburgh, Pittsburgh, PA 15260 USA
tsamard@eecs.umich.edu, ganchev@cs.pitt.edu
[2] Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48103 USA
pollackm@eecs.umich.edu

Abstract. Many systems are designed to perform both planning and execution: they include a plan deliberation component to produce plans that are then dispatched to an execution component, or *executive*, which is responsible for the performance of the actions in the plan. When the plans have temporal constraints, dispatch may be non-trivial, and the system may include a distinct *dispatcher*, which is responsible for ensuring that all temporal constraints are satisfied by the executive. Prior work on dispatch has focused on plans that can be expressed as Simple Temporal Problems (STPs). In this paper, we sketch a dispatch algorithm that is applicable to a much broader set of plans, namely those that can be cast as Disjunctive Temporal Problems (DTPs), and we identify four key properties of the algorithm.

1 Introduction

Many systems are designed to perform both planning and execution: they include a plan deliberation component to produce plans that are then dispatched to an execution component, or *executive*, which is responsible for the performance of the actions in the plan. When the plans have temporal constraints, dispatch may be non-trivial, and the system may include a distinct *dispatcher*, which is responsible for ensuring that all temporal constraints are satisfied by the executive. Prior work on plan dispatch [1-3] has focused on plans that can be represented as Simple Temporal Problems (STP) [4]. In this paper, we sketch a dispatch algorithm that is applicable to a much broader set of plans, those that can be cast as Disjunctive Temporal Problems (DTPs), and identify four key properties of the algorithm.

2 Disjunctive Temporal Problems

Definition. A **Disjunctive *Temporal Problem (DTP)*** is a constraint satisfaction problem $<V, C>$, where V is a set of variables (or nodes) whose domains are the real numbers, and C is a set of disjunctive constraints of the form C_i: $l_1 \le x_1 - y_1 \le u_1 \ \vee$

$\ldots \vee\ l_n \le x_n - y_n \le u_n$, such that for $1 \le i \le n$, x_i and y_i are both members of V, and l_i , u_i are real numbers. An ***exact solution*** to a DTP is an assignment to each variable in V satistying all the constraints in C. If a DTP has at least one exact solution, it is ***consistent***.

A DTP can be seen as encoding a collection of alternative Simple Temporal Problems (STPs). To see this, note that each constraint in a DTP is a disjunction of one or more STP-style inequalities. Let C_{ij} be the j-th disjunct of the i-the constraint of the DTP. If we select one disjunct C_{ij} from each constraint C_i, then the set of selected disjuncts forms an STP, which we will call a ***component STP*** of a given DTP. It is easy to see that a DTP D is consistent if and only if it contains at least one consistent component STP. Moreover, any solution to a consistent component STP of D is also clearly an exact solution to D itself.

Definition. A(n inexact) ***solution*** to a DTP is a consistent component STP of it. The ***solution set*** for a DTP is the set of all its solutions.

When we speak of a solution to a DTP, we shall mean an inexact solution. Plans can be cast as DTPs by including variables for the start and end points of each action.

3 A Dispatch Example

Consider a very simple example of a plan with three actions, P, Q, and R. (For presentational simplicity, we assume each action is instantaneous and thus represented by a single node). P must occur in the interval [5,10] and Q in the interval [15,20]; P and Q must be separated by at least 6 time units; and R must be performed either the interval [11,12] or [21,22]. The plan as described can be represented as the following DTP: {C1. $5 \le P - TR \le 10 \vee 15 \le P - TR \le 20$; C2. $5 \le Q - TR \le 10 \vee 15 \le Q - TR \le 20$; C3. $6 \le P - Q \le \infty \vee 6 \le Q - P \le \infty$; C4. $11 \le R - TR \le 12 \vee 21 \le R - TR \le 22$}. (Note that TR, the time reference point, denotes an arbitrary starting point.) This DTP has four (inexact) solutions: { STP_1: c_{11}, c_{22}, c_{32}, c_{41}; STP_2: c_{11}, c_{22}, c_{32}, c_{42}; STP_3: c_{12}, c_{21}, c_{31}, c_{41}; STP_4: c_{12}, c_{21}, c_{31}, c_{42}}.

Definition: An STP variable x is ***enabled*** if and only if all the events that are constrained to occur before it have already been executed. A DTP variable x is ***enabled*** if and only if it has a consistent component STP in which x is enabled.

In STP_1, both P and R are initially enabled, while in STP_3 and STP_4, Q is initially enabled. Hence, all three actions are initially enabled for the DTP. Enablement is a necessary but not sufficient condition for execution: an action must also be *live*, in the sense that the temporal constraints pertaining to its clock time of execution are satisfied. In the current example, none of the actions are initially live. The first action to become live is P, at time 5. An action is *live* during its *time window*.

Definition: The ***time window of an STP variable*** x is a pair $[l,u]$ such that $l \leq x - TR \leq u$, and for all l', u' such that $l' \leq x - TR \leq u'$, $l' \leq l$ and $u \leq u'$. Given a set of consistent component STPs for a DTP, we will write TW (x,i) to denote the time window for variable x in the i^{th} such STP. The ***upper bound*** of a time window $[l,u]$ for x in STP i, written U(x,i), is u. The ***time window of a DTP variable*** x is TW $(x)=\cup_{i\in S}$ TW (x,i), where S is the solution set of D.

The dispatcher can provide information about when actions are enabled and live in an ***Execution Table (ET).*** This is a list of ordered pairs, one for each enabled action. The first element of the entry specifies the action, and the second is a list of the convex intervals in that element's time window. For our example, then, the initial ET would be: {<*P*, {[5,10], [15,20]}>, <*Q*, {[5,10],[15,20]}}>, <*R*, {[11,12],[21,22]}>}. The ET summarizes the information in the solution STPs so that the executive does not have to handle them directly.

The ET provides information about what actions *may* be performed, but it does not provide enough information for the executive to determine what actions *must* be performed. To see this, note that the ET just given does not indicate that there is a problem with deferring both *P* and *Q* until after time 10. However, such a decision would lead to failure: if the clock time reaches 11 and neither *P* nor *Q* has been executed, then all four solutions to the DTP will have been eliminated. Thus, in addition to the information in the ET, the dispatcher must also provide a second type of information to the executive. The ***deadline formula (DF)*** provides the executive with information about the next deadline that must be met.

In the next section, we explain how to calculate the DF, which is more complicated than computing the ET. Here we simply complete the example, by illustrating how the ET and the DF would be updated as time passes. The initial DF would indicate that either *P* or *Q* must be executed by time 10. Suppose that at time 8, action *P* is executed. At this point, STP_3 and STP_4 are no longer solutions. The ET then becomes { <*Q*, {[15,20]}>, <*R*, {[11,12], [21,22]}> } and the DF is trivially "*Q* by 20" . In this case, an update to ET and DF resulted because an activity occurred. However, updates may also be required when an activity does not occur within an allowable time window. For example, if *R* has still not executed at time 13, then its entry in the ET should be updated to be just the singleton [21,22], with no changes required to the DF. The example presented in this section contains variables with very little interaction. In general, there can be significantly more interaction amongst the temporal constraints, and the DF can be arbitrarily complex.

4 The Dispatch Algorithm

We now sketch our algorithm for the dispatch of plans encoded as DTPs. The input is a DTP and the output is an Execution Table (ET) and a Deadline Formula (DF). For each pair <*x*, TW(*x*)> in ET, *x* must be executed some time within TW(*x*). It is up to the executive to decide exactly when. The DF imposes the constraint that *F* has to

hold by time *t*, where a variable that appears in the DF becomes true when its corresponding event is executed.

The dispatch algorithm will be called in three circumstances: (1) when a new plan needs to have its dispatch information initialized, at or before time *TR*; (2) when an event in the DTP is executed; (3) when an opportunity for execution passes because the clock time passes the upper bound of a convex interval in the time window for an action that has not yet been executed. Pseudo-code is provided in Figure 1. Space constraints preclude detailed description of the algorithm (but see [5]). Here we simply illustrate the procedure for computing the DF, the most interesting part of the algorithm.

Recall the example above. Initially, at time TR, the DTP has four solutions. To determine the initial DF, we consider the next critical moment, NC, which is the next time at which any action must be performed. This time is equal to the minimal value of all the upper bounds on time windows for actions, i.e., it is min{U(x,i)| x is an action in the DTP, and i is a solution STP}. For instance, in our example DTP, U(P, 1) = U(P, 2) = 10. The actions that may need to be executed by NC are those x such that U(x,i) = NC for some STP i. We create a list UMIN containing ordered pairs <x,i> such that U(x,i) = NC. In our current example, UMIN = {<P, 1>, <P, 2>, <Q, 3>, <Q, 4>}. Now we perform the interesting part of the computation. If <x,i> is in UMIN , it means that unless x is executed by time NC, STPi will cease to be a solution for the DTP. It is acceptable for STPi to be eliminated from the solution set only if there is at least one alternative STP that is not simultaneously eliminated. This is exactly what the deadline formula ensures: that at the next critical moment, the entire set of solutions will not be simultaneously eliminated. We thus use a minimal set cover algorithm to compute all sets of pairs <x,i> in UMIN such that the i values form a minimal cover of the set of solution STPs. In our example, there is only one minimal cover, namely the entire set UMIN. Thus, the initial DF specifies that P or Q must be executed by time 10: <P∨ Q, 10>. In general, there may be multiple minimal covers of the solution STPs: in that case, each cover specifies a disjunction of actions that must be performed by the next critical time. For instance, suppose that some DTP has four solution STPs, and that at time TR, U (L, 1) = U (L, 2) = U (M, 3) = U (M, 4) = U (N, 4) = U (S, 3) = 10. Then by time 10 either L or M must be executed; additionally, at least one of L or N or S must be executed. The corresponding DF is <(L∨ M)∧(L∨ N∨ S), 10>.

5 Formal Properties of the Algorithm

The role of a dispatcher is to notify the executive of when actions may be executed and when they must be executed. Informally, we will say that a dispatch algorithm is ***correct*** if, whenever the executive executes actions according to the dispatch notifications, the performance of those actions respects the temporal constraints of the underlying plan. Obviously, dispatch algorithms should be correct, but correctness is not enough. Dispatchers should also be ***deadlock-free*****:** they should provide enough information so that the executive does not violate a constraint through inaction. A

Initial-Dispatch (DTP D)

1. Find all n solutions (consistent component STPs) to D, calculate their distance graphs, and store them in Solutions [i]. Associate each solution with its (integer-valued) index.
2. Set the variable TR to have the status Executed, and assign TR=0.
3. Compute-Dispatch-Info(Solutions).

Update-for-Executed-Event (STP [i] Solutions)

1. Let x be the event that was just executed, at time t.
2. Remove from Solutions all STPs i for which $t \notin$ TW (x,i).
3. Propagate the constraint $t \leq x - TR \leq t$ in all remaining Solutions.
4. Mark x as Executed.
5. Compute-Dispatch-Info (Solutions).

Update-for-Violated-Bounds (STP[i] Solutions)

1. Let $U = \{U(x, k) | U(x, k) < Current\text{-}Time\}$
2. Remove from Solutions all STPs k that appear in U.
3. Compute-Dispatch-Info (Solutions).

Compute-Dispatch-Info (STP[i] Solutions)

1. For each event x in Solutions
2. {If x is enabled
3. ET = ET $\cup$ <x, TW(x)>}.
4. Let U = the set of upper bounds on time windows, U(x,i) for each still un-executed action x and each STP i.
5. Let NC, the next critical time point, be the value of the minimum upper bound in U.
6. Let $U_{MIN} = \{U(x, i) | U(x,i) = NC\}$.
7. For each x such that $U(x,i) \in U_{MIN}$, let $S_x = \{i \mid U(x,i) \in U_{MIN}\}$
8. {Initialize F = true;
9. For each minimal solution MinCover of the set-cover problem (Solutions, $\cup S_x$), let $F = F \wedge (\vee_{x \mid Sx \in MinCover} x)$.
10. DF = <F, NC>.}

Figure 1. The Dispatch Algorithm

third desirable property for dispatchers is ***maximal flexibility:*** they should not issue a notification that unnecessarily eliminates a possible execution, i.e., an execution that respects the constraints of the underlying plan. Finally, we will require dispatch algorithms to be ***useful***, in the sense that they really do some work. Usefulness will be defined as producing outputs that require only polynomial-time reasoning on the part of the executive. Without a requirement of usefulness, one could achieve the other three properties by designing a DTP dispatcher that simply passed the DTP representation of a plan on to the executive, letting it do all the reasoning about when to execute actions.

Our dispatch algorithm has these four properties, as proved in [5]. The proofs depend on a more precise notion of how the dispatcher and the executive interact. The dispatcher issues a *notification sequence*, a list of pairs $<ET,DF>_1 \ldots ,<ET,DF>_n$, with a new notification issued every time an event is executed or an upper bound is passed. The executive performs an *execution sequence*, a list $x_1= t_1, \ldots, x_n=t_n$ indicating that event x_i is executed at time t_i, subject to the restriction that $j>i \Rightarrow t_j > t_i$. An execution sequence is complete if it includes an assignment for each event in the original DTP; otherwise it is partial. The notification and execution sequences will be interleaved in an *event sequence*. We associate each execution event with the preceding notification, writing Notif(x_i) to denote the notification of event x_i.

Definition. An execution sequence *E respects* a notification sequence *N iff*

1. For each execution event $x_i=t_i$ in *E*, $<x_i$, TW $(x_i)>$ appears in ET of Notif (x_i) and $t_i \in$ TW(x_i), i.e., each event is performed in its allowable time window.
2. For each DF=$<F,t>$ in *N*, $\{x_i \,|x_i = t_i \in E$ and $t_i \leq t\}$ satisfies *F*. That is, the execution sequence satisfies all the deadline formulae.

Theorem 1: The dispatch algorithm in Fig. 1 is correct, i.e., any complete execution sequence that respects its notifications also satisfies the constraints of D.
Theorem 2: The dispatch algorithm in Fig. 1 is deadlock-free, i.e., any partial execution that respects its notifications can be extended to a complete execution that satisfies the constraints of D.
Theorem 3: The dispatch algorithm in Fig. 1 is maximally flexible, i.e., every complete execution sequence that respects the constraints in *D* will be part of some complete event sequence.
Theorem 4: The dispatch algorithm in Fig. 1 is useful, i.e., generating an execution sequence is polynomial in the size of the notifications.

References

1. Muscettola, N., P. Morris, and I. Tsamardinos. Reformulating Temporal Plans for Efficient Execution. in Proceedings of the 6th Conference on Principles of Knowledge Representation and Reasoning. 1998.
2. Tsamardinos, I., P. Morris, and N. Muscettola, Fast Transformation of Temporal Plans for Efficient Execution, in Proceedings of the 15th National Conference on Artificial Intelligence. 1988, AAAI Press/MIT Press: Menlo Park, CA. p. 254-261.
3. Wallace, R.J. and E.C. Freuder, Dispatchable Execution of Schedules Involving Consumable Resources, in Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling. 2000.
4. Dechter, R., I. Meiri, and J. Pearl, Temporal Constraint Networks. Artificial Intelligence, 1991. 49: p. 61-95.
5. Tsamardinos, I., Constraint-Based Temporal Reasoning Algorithms, with Applications to Planning. 2001.

Optimising Plans using Genetic Programming

C Henrik Westerberg and John Levine

CISA, University of Edinburgh,
80 South Bridge, Edinburgh, EH1 1HN
{carlw,johnl}@dai.ed.ac.uk
http://www.dai.ed.ac.uk/homes/carlw/

Abstract. Finding the shortest plan for a given planning problem is extremely hard. We present a domain independent approach for plan optimisation based on Genetic Programming. The algorithm is seeded with correct plans created by hand-encoded heuristic policy sets. The plans are very unlikely to be optimal but are created quickly. The sub-optimal plans are then evolved using a generational algorithm towards the optimal plan. We present initial results from Blocks World and found that GP method almost always improved sub-optimal plans, often drastically.

1 Introduction

Finding any plan for planning domains is often a difficult task, but we are often more interested in the even harder task of finding optimal or near optimal plans. The current fastest planning systems use heuristics and hill-climbing techniques. However, no heuristic is perfect and plans found in this way are often sub-optimal, in the sense they use more actions to achieve the goal state than are necessary.

We present a domain independent technique, based on Genetic Programming (GP) that attempts to optimise linear plans. The system accepts a seed of plans from which to optimise. This seed could be produced by a current planning system or plans made using heuristics. The amount of computational effort to devote to the optimisation stage can also be set by the user by setting various parameters of the system. The GP algorithm also has *anytime* behaviour, and could return the best current plan at any time during the run.

Using the Genetic Planning optimisation system, we experimented on two domains: Blocks World Domain, and the Briefcase Domain [6]. The Blocks World problems were kindly donated to us Jose Ambité. The results of the Briefcase Domain have been omitted due to space restrictions. During the experimentation we were looking for how much the initial plans could be shrunk depending on the type of heuristics used, the behaviour of the system as it operated, and what changes we could make to the current system to improve its ability.

2 Plan Optimisation via Genetic Programming

We present here one possible implementation of using Genetic Programming as a linear plan optimiser. We used two different hand-encoded *policy sets* for the Blocks Domain in order to seed the initial population with correct but overly long plans. We then used a generational algorithm with standard genetic operators in order to optimise those plans [2]. We based our work on a previously implemented generational algorithm for linear planning [8].

The following implementational details have many alternatives, and are not fixed. One of the strengths of our approach is that the Fitness Function and Simulation stage can be altered to look for different cost aspects besides the simplistic plan length.

Plan Representation: Plans are represented as linear lists of sequential, instantiated, atomic actions. Each atomic action contains one operator and its arguments.

Simulation: The simulation stage takes an individual or plan and then attempts to apply all the actions. During the simulation stage various attributes of the plan can be recorded such as how many actions there are in the plan and what effect the plan had on the initial state. This information can then be used as input by the fitness function.

Fitness Function: The fitness function takes the output of the simulation stage and prescribes a fitness value to individual based on the information given to it. In the case of this system the fitness function has two parts. The first part says whether the plan achieves all the goals not. The second part is the number of actions in the plan and is used as a tie-breaker in tournament selection.

2.1 Genetic Operators

There is a large choice of genetic operators to be used during the optimisation stage. We have taken the position of keeping things simple and stochastic. One alternative is to implement domain specific operations, such as rewrite rules, for optimising particular domains.

Crossover: This system implements 1-point crossover.

Reproduction: This is the simplest operator and it copies the selected parent into the next population.

Shrink Mutation: This type of mutation simply deletes a randomly selected action from the parent.

Move Mutation: This type of mutation moves a randomly selected action to a new randomly selected position.

Mutations occur on children created by either reproduction or crossover. The probabilities of the operators occuring are set by the user. The implementation presented here is based on an existing system and is by no means optimal for generating optimal plans. Improvements that can be made to it and some are suggested in Section 5.

3 Policy Set Planning in Blocks World

The Blocks World Domain is important because it is one of the benchmarking domains used to compare different planners. Blocks is also important historically as one of the original planning problem domains. In addition, finding optimal plans for Blocks World problems is known to be NP-hard [3]. We also chose the Blocks World Domain as a fast domain specific planning algorithm that produces optimal plans exists for it, called BWOPT [7].

Our system uses hand-encoded policy sets to produce entire populations filled with correct but suboptimal plans. A GP optimising system probably works better if the initial populations is diverse. To achieve this the policy sets were interpreted non-deterministically.

The policy sets generally function like this. The rules within each policy set are tested sequentially. For the current rule the current world state is examined and all actions that could operate on that state in accordance with the rule are discovered. At that point, one of the actions is selected randomly and added to the new plan. The current world state is updated and the formation of the plan continues until all goals in the goal state are achieved. If the rule allows for no actions, the next rule in the rule set is used and if one rule fires then the other rules are ignored.

There are several types of policy sets that can characterised by how easy it is to optimise the resulting plan. The three types we are interested in here are:

- **Optimal Policy Sets:** These policy sets always produce optimal plans, for any problem in the domain.
- $\mathcal{DM}$**-Optimal Policy Sets:** These policy sets always produce plans where the optimal plan can be discovered by only deleting and moving actions:
  - $\forall c \in C \rightarrow c^* \in \mathcal{DM}(c)$ where C is the set of all plans constructed by the policy set, c^* is the optimal plan and $\mathcal{DM}(c)$ is the set of all plans which can be created by only moving and deleting actions in c.
- **Satisficing Policy Sets:** These policy sets produce correct plans but may produce plans that are missing actions which the optimal plan would need.

Policy Set 1

1. Discover all actions achieving well placed blocks *or*
2. Find all actions moving movable non-well placed blocks to a new location

Policy Set 2

1. Discover all actions placing movable blocks onto the table *then*
2. Discover all actions achieving well placed blocks

A well placed block is one which no longer has to move, as it is in its target location and all blocks below it are well placed. A movable block is one which is not underneath a block or already on the table. Policy Set 1 does not scale very well for larger problem instances: when the first rule provides no actions,

it "wanders" around at random until the first rule starts to succeed. The first policy set belongs in the class of *Satisficing Policy Sets*. The second policy set unstacks all the blocks and then stacks the blocks back up in the right order. This policy set belongs in the class of $\mathcal{DM}$*-Optimal Policy Sets*. This was the policy set Ambité used in PbR [1].

4 Experimental Results

Each experiment was done using the parameters shown in Table 1. We performed 25 runs for each problem, and again for each policy set. We experimented using 50 Blocks World problems. During the run we recorded the average number of actions in the first best individual (from the seeding stage) and the average number of actions in the last best individual.[1] Each point on the x-axis represents a single problem. The order of the problems is first by blocks size, and then by average length of the first best individual.

Parameter	Setting
Termination	Maximum number of generations is 1000
Population Size	20 plans
Initial Length	Maximum 400 actions
Tournament Size	2
Maximum Plan Size	1000 actions
Genetic Operators	5% crossover and 95% reproduction
Shrink Mutation	Applied to 5% of children, 1 delete
Move Mutation	Applied to 5% of children, 1 move

Referring to Figure 1, Policy Set 1 shows significant but not complete improvement in plan length after 1000 generations. An additional termination criterion was implemented, called "no change" which stops a run if there is no change in fitness after X generations. We repeated the experiments setting X to 5000. Taking the 30 block problems as an example, these were shrunk down to the 50 action mark.

Referring to Figure 2, some improvement could be made to the initial plan within 1000 generations even though the initial plans were reasonably close to optimal. The no change results managed to to shrink the plans a little more, and taking the 30 block problems again, these were shrunk down to around the 40 action mark. This difference between the two policy sets is returned to in the conclusions.

Also included in Figure 2 are results from FF [4]. We ran the 3 plans produced for the 30 block problems using the no change setup. The results are indicated with the triangles, and show significant shrinkage.

[1] CPU times are not considered as the system was implemented using Java, and running on Solaris. System times can be dramatically improved if written for C under Linux.

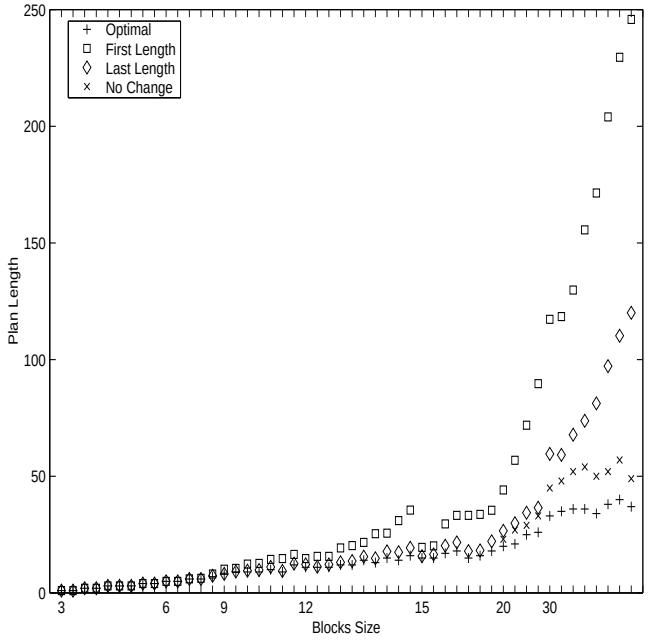


Fig. 1. Policy Set 1 on the Blocks World Problems

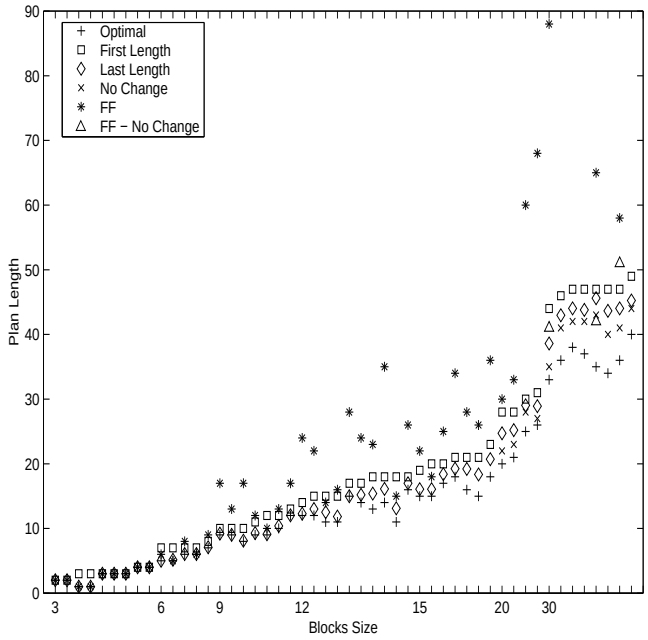


Fig. 2. Policy Set 2 on the Blocks World Problems

5 Conclusions and Future Work

The most successful current planners use heuristics and hill-climbing techniques. However, since no heuristic is perfect, such techniques often produce suboptimal plans, as in the case of FF. We have presented a linear plan optimisation technique, based on GP, which attempts to optimise plans. The system is domain independent, and can be used as addition to existing linear plan synthesisers. The system uses simple operations like mutation and crossover in order to accomplish this. The system could optimise plans to varying degrees of success depending on where the plans came from. A tentative conclusion is that plans made by $\mathcal{DM}$-Optimal policy sets can be optimised further towards the shortest plan than those made by satisficing policy sets.

We want to improve on the Generational framework suggested here for plan optimisation. There are a number of alternatives, such as a steady state algorithm, that we could adopt to decrease the length of the resulting plans. Also the system could be redesigned to optimise single plans.

We also want to broaden the definition of optimal to mean more than just plan length. More complicated domains with time, plan execution by an agent, resources, and so on, would make plan optimisation a multi-dimensional problem. It seems plausible that a genetic technique would be suitable for this kind of optimisation due to the way fitness functions and simulation are used.

References

1. Ambité, J.L., Knoblock, C.A.: Planning by Rewriting: Efficiently Generating High-Quality Plans. In Proceedings of the 14th National Conference on Artificial Intelligence, Providence RI USA, 1997
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction. San Francisco CA, Morgan Kaufmann Publishers, 1998
3. Bylander, T.: The computational complexity of propositional STRIPS planning. In Journal of Artificial Intelligence, 69(1-2):165-204, 1994
4. Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search. In Journal of Artificial Intelligence Research, Volume 14, Pages 253-302, 2001
5. Koza, J. R.: Genetic Programming. The MIT Press, Cambridge MA USA, 1992
6. Muslea, I.: SINERGY: A Linear Planner Based on Genetic Programming. In Proceedings of the 4th European Conference on Planning, pages 312-324, Toulouse France, Springer, Sep 1997
7. Slaney, J., Thiébaux, S.: BLOCKS WORLD TAMED Ten thousand blocks in under a second. Technical Report TR-ARP-17-95, Automated Reasoning Project, Australian National University, Oct 1995
8. Westerberg, C.H., Levine, J.: "GenPlan": Combining Genetic Programming and Planning. In Proceedings for the UK Planning and Scheduling Special Interest Group, Milton Keynes UK, Dec 2000
9. Westerberg, C.H., Levine, J.: Investigation of Different Seeding Strategies in a Genetic Planner. In Applications of Evolutionary Computing, Proceedings of EuroGP, pages 505-514, Lake Como Italy, Springer, Apr 2001

Demo Papers

A Demonstration of Robust Planning, Scheduling and Execution for the Techsat-21 Autonomous Sciencecraft Constellation

Steve Chien, Rob Sherwood, Michael Burl, Russell Knight, Gregg Rabideau[1], **Barbara Engelhardt, Ashley Davies, Rebecca Castano, Tim Stough, Joe Roden** [1] **Paul Zetocha, Ross Wainwright, Pete Klupar,** [2] **Pat Cappelaere, Jim Van Gaasbeck,** [3] **Derek Surka, Margarita Brito,** [4] **Brian Williams, Mitch Ingham**[5]

[1]Jet Propulsion Laboratory, California Institute of Technology
[2]Air Force Research Laboratory
[3]Interface & Control Systems
[4]Princeton Satellite Systems
[5]Space Systems Laboratory, Massachusetts Institute of Technology

Abstract. The Autonomous Sciencecraft Constellation (ASC) will fly onboard the Air Force's TechSat-21 constellation (scheduled for launch in 2004). ASC will use onboard science analysis, replanning, robust execution, model-based estimation and control, and formation flying to radically increase science return by enabling intelligent downlink selection and autonomous retargeting. These capabilities will enable tremendous new science that would be unreachable without this technology. We offer a demonstration of the planning, scheduling, and execution framework used in ASC.

1 Context

Robust planning, scheduling, and execution in a real environment is a challenging task. In general, each of these elements is difficult in their own right, and the fusion of these can be equally challenging. We offer a demonstration of the integration of each of these in the real task of operating the Techsat-21 (TS21) constellation of sciencecraft [4]. Our demonstration includes nominal operations as well as operations with anomalies. Our system is capable of generating its own science goals based on previous information-gathering activities. In general, the system as a whole provides considerable autonomy. The rest of this document describes the specific mission and its background, as well as our approach to addressing the challenges we face in flying such a mission. ·

TechSat-21 is scheduled for a late 2004 launch and will fly three satellites in a near circular orbit at an altitude of 600 Km. The primary mission is one-year in length with the possibility for an extended mission of one or more additional years. One of the objectives of TechSat-21 is to demonstrate advanced radar systems. The principal processor onboard each of the three TechSat-21 spacecraft is a BAE Radiation hardened 175 MIPS, 133MHz PowerPC 750 running the OSE 4.3 operating system.

· Portions of this work were performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Other portions were performed at the Massachusetts Institute of Technology Space Systems and Artificial Intelligence Laboratories, under contracts from AFOSR and the DARPA Mobies program.

OSE was chosen because it is message passing and thus suitable for distributed applications. Each satellite will have 128 Mbytes of SDRAM as well as considerable (Gigabytes) of disk storage

2 Autonomy Technologies and Scenario

The ASC onboard flight software includes several autonomy software components:

1. *Onboard science algorithms* [1] that will analyze the image data, generate derived science products, and detect trigger conditions such as science events, "interesting" features, and change relative to previous observations
2. *Model-based mode identification and execution (MI-R)* that uses component-based hardware models to analyze anomalous situations and to generate novel command sequences and repairs.
3. *Robust execution management software* using the Spacecraft Command Language (SCL) package to enable event-driven processing and low-level autonomy
4. The Continuous Activity Planning, Scheduling, and Replanning (CASPER) *planner* that will replan activities, including downlink, based on science observations in the previous orbit cycles
5. The ObjectAgent and TeamAgent *cluster management software* will enable the three Techsat-21 spacecraft to autonomously perform maneuvers and high precision formation flying to form a single virtual instrument

We will demonstrate ASC – specifically autonomous recognition of science events and response including planning and execution. For example, ASC will monitor lava flows in Hawaii and respond as follows:

1. Initially, ASC has a list of science targets to monitor.
2. As part of normal operations, CASPER generates a plan to monitor the targets on this list by periodically imaging them with the radar.
3. During such a plan, a spacecraft images the volcano with its radar.
4. The *Onboard Science Software* compares the new image with previous image and detects that the lava field has changed due to new flow. Based on this change the science software generates a goal to acquire a new high-resolution image of an area centered on the new flow.
5. The addition of this goal to the current goal set triggers the CASPER planner to modify the current operations plan to include numerous new activities in order to enable the new science observation. During this process CASPER interacts with ObjectAgent to plan required slews and/or maneuvers.
6. SCL executes this plan in conjunction with several autonomy elements. Mode Identification assists by continuously providing an up to date picture of system state. Reconfiguration (Burton) achieves configurations requested by SCL. And ObjectAgent and TeamAgent execute maneuvers planned by CASPER and requested at run-time by SCL.
7. Based on the science priority, imagery of identified "new flow" areas; are downlinked. This science priority could have been determined at the original event detection based on subsequent onboard science analysis of the new image.

As demonstrated by this scenario, onboard science processing and spacecraft autonomy enable the focus of mission resources onto science events so that the most interesting science data is downlinked. In this case, a large number of high priority

science targets can be monitored and only the most interesting science data (during times of change and focused on the areas of change) need be downlinked.

3 Planning and Execution Framework

ASC utilizes a hierarchical architecture for planning and execution. CASPER operates at the highest level of abstraction, creating mission plans in response to high level science and engineering goals. CASPER performs traditional planning and scheduling and reasons about high level states and resources. CASPER uses local search, iterative repair, and continuous planning to respond at the 10s of seconds timescale (for further details see [2]). CASPER is being deployed in a wide range of applications including spacecraft operation, rover control, ground communications station automation, and high level control of unpiloted aerial vehicles.

Scheduled CASPER activities correspond to scripts in the Spacecraft Command Language (SCL) [10] that provide robust execution. SCL integrates procedural programming with a real-time, forward-chaining, rule-based system to provide a "smart" executive command and control function. This functionality can be used to implement retries use of alternate execution methods for robust execution as well as fault detection, isolation and recovery (FDIR). SCL is a mature software product, and has successfully flown on Clementine I and ROMPS.

Mode identification (MI) uses a declarative model to interpret sensor information to determine the configuration of the system in the presence of incomplete, noisy data. Mode reconfiguration uses these same models to determine command sequences to achieve desired configurations. The executive uses the model to track planner goals, confirm hardware modes, reconfigure hardware, generate command sequences, detect anomalies, isolate faults, diagnose, and perform repairs.

Our model-based execution framework is an enhanced version of the Burton system, described in [9] and under development at the MIT AI and Space Systems laboratories. This framework incorporates the Mode Estimation capabilities of Livingstone 1 and 2, described in [5] and developed at NASA Ames. The marriage between the model-based executive and SCL provides a powerful hybrid execution capability with an expressive scripting language and an extensive capability to generate novel responses to anomalous situations.

ObjectAgent and TeamAgent [7] provide an autonomous maneuver and formation flying capability for ASC. At plan time, CASPER consults OA and TA on the feasibility and resource requirements to perform formation changes, maneuvers, and slews. At execution time, formation changes, maneuvers, and slews planned by CASPER and requested by SCL are performed by OA and TA. In this execution time function OA and TA perform closed loop control in their use of lower-level attitude and control software to achieve the desired goals.

4 Related Work and Conclusions

In 1999, the Remote Agent experiment (RAX) [6] executed for several days onboard the NASA Deep Space One mission. RAX demonstrated a batch onboard planning capability but did not demonstrate onboard science. RAX also included an earlier version of the Livingstone and Burton mode identification and fault recovery software. PROBA[8] is a European Space Agency (ESA) mission launching in 2001 that will be demonstrating onboard autonomy.

The Three Corner Sat (3CS) University Nanosat mission will be using the CASPER onboard planning software integrated with the SCL ground and flight

execution software [3]. Launching in 2002, 3CS will use onboard science data validation, replanning, robust execution, and multiple model-based anomaly detection. The 3CS mission is considerably less complex than Techsat-21 but still represents an important step in the integration and flight of onboard autonomy software.

ASC will fly on the Techsat-21 mission will demonstrate an integrated autonomous mission using onboard science analysis, replanning, robust execution, model-based estimation and control, and formation flying. ASC will perform intelligent science data selection that will lead to a reduction in data downlink. In addition, the ASC experiment will increase science return through autonomous retargeting. Demonstration of these capabilities in onboard the Techsat-21 constellation mission will enable radically different missions with significant onboard decision-making leading to novel science opportunities. The paradigm shift toward highly autonomous spacecraft will enable future NASA missions to achieve significantly greater science returns with reduced risk and cost.

5 References

[1] M.C. Burl, W.J. Merline, E.B. Bierhaus, W. Colwell, C.R. Chapman, "Automated Detection of Craters and Other Geological Features *Intl Symp Artificial Intelligence Robotics & Automation in Space*, Montreal, Canada, June 2001

[2] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Using Iterative Repair to Improve Responsiveness of Planning and Scheduling," *Proc Fifth Intl Conf on Artificial Intelligence Planning and Scheduling,* Breckenridge, CO, April 2000.

[3] S. Chien, B. Engelhardt, R. Knight, G. Rabideau, R. Sherwood, E. Hansen, A. Ortiviz, C. Wilklow, S. Wichman " Onboard Autonomy on the Three Corner Sat Mission," *Intl Symposium on Artificial Intelligence Robotics and Automation in Space*, Montreal, Canada, June 2001

[4] S. Chien, R. Sherwood, M. Burl, R. Knight, G. Rabideau, B. Engelhardt, A. Davies, P. Zetocha, R. Wainright, P. Klupar, P. Cappelaere, D. Surka, B. Williams, R. Greeley, V. Baker, J. Doan, "The Techsat-21 Autonomous Sciencecraft Constellation", *ProcIntl Symp Artificial Intelligence Robotics and Automation in Space*, Montreal, Canada, June 2001

[5] J. Kurien and P.P. Nayak, "Back to the Future for Consistency-based Trajectory Tracking," *Proc. National Conference on Artificial Intelligence*, Austin, TX, 2000.

[6] rax.arc.nasa.gov, Remote Agent Experiment Home Page.

[7] D. Surka, J. Mueller, M. Brito, B. Urea, M Campbell, "Agent-based Control of Multiple Satellite Formation Flying," *Intl Symposium on Artificial Intelligence Robotics and Automation in Space*, Montreal, Canada, June 2001

[8] The PROBA Onboard Autonomy Platform, http://www.estec.esa.nl/proba/

[9] B.C. Williams and P.P. Nayak, "A Model-based Approach to Reactive Self-Configuring Systems," *Proceedings of the National Conference on Artificial Intelligence*, Portland, Oregon, 1996.

[10] Spacecraft Command Language, www.interfacecontrol.com

DISCOPLAN: an Efficient On-line System for Computing Planning Domain Invariants*

Alfonso Gerevini[1] **Lenhart Schubert**[2]

[1] Dipartimento di Elettronica per l'Automazione, Università di Brescia
Via Branze 38, 25123 Brescia, Italy. E-mail: gerevini@ing.unibs.it

[2] Department of Computer Science, University of Rochester
Rochester, NY 14627-0226. E-mail: schubert@cs.rochester.edu

Abstract

DISCOPLAN is an efficient system for discovering state invariants in planning domains with conditional effects. Among the types of invariants found are implicative constraints relating a fluent predication to a fluent or static predication (with allowance for static supplementary conditions), single-valuedness constraints, exclusiveness constraints, and several others. The algorithms used are polynomial-time for any fixed bound on the number of literals in an invariant. Some combinations of constraints are found by simultaneous induction, and the methods can be iterated by expanding operators using previously found invariants. The invariants found by DISCOPLAN have been shown to enable large performance gains in SAT planners, and they can also be helpful in planning domain development and debugging.

Introduction

State invariants (or *state constraints*) in planning are properties of objects or relationships among objects that hold in all states reachable from the initial state. For example, a familiar invariant in a blocks world is the property that that if one block is on another, the latter is not clear. In our terminology, this is an *implicative* constraint. Another example is that a block can be on at most one other block; this is a *single-valuedness* constraint (sv-constraint).

A point that has become widely recognized in the planning community (and that we amplify in what follows) is that knowledge of state invariants is important for efficient planning. However, such knowledge cannot in general be assumed to be available *a priori* in a given planning domain. Rather, planning domains are generally considered fully specified once a set of operators with well-defined preconditions and effects has been supplied, along with an initial state. This is defensible since state invariants are implicit in the specification of the operators and initial state; i.e., under a STRIPS assumption the only properties and relationships that change when an operator is applied are those spelled out in the effects of the operator. So a separate specification of what remains unchanged when operators are applied would be logically redundant. However, it is far from obvious from inspection of a given set of planning operators and an initial state what the invariants of the domain are. The goal of our research has been to formulate automatic, efficient methods for inferring the most important such invariants, and to implement these methods in our DISCOPLAN system.

*The on-line DISCOPLAN system can be accessed at `http://prometeo.ing.unibs.it/discoplan`. DISCOPLAN is written in Common Lisp.

The importance of state invariants for efficient planning is that they can be used to radically restrict the search space. This is so for any approach to planning that involves explicit or implicit exploration of incompletely specified possible states of the world, as is the case for deductive planning, regression planning, bidirectional planning, and planning by incremental constraint satisfaction (in particular, SAT-based planning).

In our work we have focused on SAT-based planners. These implicitly search a space of state sequences, constrained by disjunctions of ground literals. Their performance depends critically on the invariants added (as ground instances) to the mix of disjunctions, and intuitively this is because state invariants constrain the alternative states that are possible at each time step under consideration. Some results showing the dramatic improvements in the performance of SAT-based planners like SATPLAN [8] and MEDIC [2] obtainable through the use of automatically inferred invariants are included in [5].

DISCOPLAN finds a variety of different types of constraints, including static (type) constraints (most importantly, supertype / subtype and exclusion relations among static monadic predicates – ones unaffected by any operator), and predicate domain constraints (sets of possible argument tuples corresponding to each predicate in the domain, after 0, 1, ..., t actions have occurred). But the majority of its algorithms are devoted to the discovery of state invariants, using a *hypothesize-and-test* paradigm. All the algorithms instantiating this paradigm are applicable to sets of operators conforming with UCPOP or PDDL syntax [11, 7], allowing for *when*-clauses (conditional effects) but not disjunctive or universally quantified conditions. We will be referring to the unconditional part of an operator as its *primary when-clause*. The allowance for conditional effects is a major distinction of DISCOPLAN from related systems.

Very briefly, the hypothesize-and-test paradigm consists of hypothesizing invariants Γ of some particular syntactic type, such as implicative constraints (`IMPLIES` ϕ ψ) where ϕ and ψ are literals that may contain universal variables, augmenting these hypotheses with potential supplementary static conditions, and then testing them against all *when*-clauses of all operators and against the given initial conditions. In the testing phase, minimal sets Σ of supplementary conditions are found, up to sets of some limited size (e.g., 2 or 3) that suffice to ensure that $\Sigma \Rightarrow \Gamma$ holds in all states reachable from the initial state. The hypothetical invariants Γ of a particular type are chosen by inspecting the preconditions and effects of particular operators, to find conditions that appear to become or remain true when

certain kinds of effects occur. The idea is to choose the constituents of Γ in such a way that a proof by induction of the invariance of Γ will be at least locally enabled. In this way large numbers of syntactically possible invariants are eliminated from consideration. The testing phase can be viewed as an automated inductive proof attempt (with addition of supplementary conditions as needed to allow the proof to succeed). An important point is that Γ may actually consist of multiple hypotheses that can be proved to be invariants by simultaneous induction. Typically, such multiple hypotheses consist of an implicative hypothesis (`IMPLIES` ϕ ψ) along with sv-hypotheses corresponding to argument positions in ϕ and ψ occupied by universal variables occurring in only one of ϕ, ψ. The point is important since the invariance of the individual formulas in such cases cannot be proved in isolation. Our various hypothesize-and-test algorithms have been proved to yield correct invariants, and run in polynomial time for fixed bound on the number of supplementary conditions Σ added to Γ.

In a little more detail, the hypothesize-and-test algorithms conform with the following structure (iterating over all possible candidate constraints Γ found in the first step).

1. Hypothesize a constraint Γ based on co-occurrences of literals in a *when*-clause w of an operator and in the corresponding primary *when*-clause w_1 (if different). For example, effects ϕ and ψ might lead to an implicative hypothesis (`IMPLIES` ϕ ψ), and possibly sv-hypotheses about the predicates involved.
2. Add a set of candidate supplementary conditions $\{\sigma_1, ..., \sigma_n\}$, consisting of the static preconditions of w and w_1 and if $w \neq w_1$, the negations of static preconditions of other *when*-clauses (except ones that unify with static preconditions of w or w_1 or their negations).
3. Test hypothesis Γ relative to each *when*-clause of each operator, using the relevant *verification conditions*; for each apparent violation of Γ find the corresponding possible "excuses" for the violation. An excuse is a set of provisos $\{\sigma'_1, ..., \sigma'_m\}$, chosen from the candidate supplementary conditions, that weaken the hypothesis sufficiently to maintain its truth. If a violation has no excuses, abandon the hypothesis Γ, otherwise record the set of possible excuses of the violation on a global list.
4. Find all minimal subsets (up to a given size, e.g., 3) of $\{\sigma_1, ..., \sigma_n\}$ that "cover" all apparent violations of Γ; a subset of $\{\sigma_1, ..., \sigma_n\}$ covers an apparent violation of Γ if it contains all elements of at least one "excuse" for that violation;
5. Check hypothesis (Γ $\sigma'_1 ..., \sigma'_m$) (i.e., the original hypothesis together with added provisos) for each of the minimal subsets $\{\sigma'_1, ..., \sigma'_m\}$ of $\{\sigma_1, ..., \sigma_n\}$ found in the previous step for truth in the initial conditions of the problem being solved; return the variant hypotheses that pass this test as the verified hypotheses.

The *verification conditions* referred to in step 3 depend on the form of Γ, and are designed to ensure that if Γ together with specified supplementary conditions holds in a given state, it also holds in every possible successor state. For example, in the case of a simple implicative constraint (`IMPLIES` ϕ ψ) together with a set of static supplementary conditions, the verification conditions say (roughly) that any operator effect matching ϕ must be accompanied by an effect or persistent precondition matching ψ, or else the preconditions must entail the falsity of a supplementary condition; and similarly for the contrapositive, (`IMPLIES` $\neg\psi$ $\neg\phi$). (The conditions are actually slightly more complicated because of the allowance for conditional effects.)

```
(define (operator Put)
  :parameters (?x ?y ?z)
  :precondition (and (on ?x ?z) (clear ?x)
                     (neq ?x Table) (neq ?y ?z) (neq ?x ?y))
  :effect (and (when (eq ?y Table)
                  (and (on ?x ?y) (clear ?z) (not (on ?x ?z))))
               (when (and (neq ?y Table) (clear ?y))
                  (and (on ?x ?y) (clear ?z) (not (on ?x ?z))
                       (not (clear ?y))))) )
```

Figure 1: A Formalization of the blocks world.

Types of DISCOPLAN Invariants

The input of DISCOPLAN is a domain description consisting of the specification of an initial state and a set of extended STRIPS operators which may involve conditional effects, negated preconditions, constants, typed and untyped parameters (Figure 1 gives a very simple formalization of the blocks world containing some of these fetures). In the following we describe the types of invariants that are discovered by the current version of DISCOPLAN (for a more detailed description the reader is referred to [5, 6]).

Predicate Domain Constraints. Predicate domain constraints are sets of possible argument tuples corresponding to each predicate in the domain after 0, 1, ..., t actions have occurred. These constraints are computed using a simplified version of the planning graph for the given problem [1].

Static Predicates and Static Constraints. Static constraints are invariants involving type-predicates, i.e., static monadic predicates that occur positively in the initial state – ones unaffected by any operator. Static constraints consist of a (possibly empty) set of objects for each type-predicate and a list of supertype, subtype, and incompatible relationships between type-predicates.

Simple Implicative Constraints. Simple Implicative Constraints are constraints of form $((\phi \Rightarrow \psi)\ \sigma_1 ... \sigma_k)$, where ϕ, ψ, and $\sigma_1, ..., \sigma_k$ are function-free *literals*, i.e., negated or unnegated atomic formulas whose arguments are constants or variables. Such constraints are to be interpreted as saying "In every state, for all values of the variables, if ϕ then ψ, *provided that* σ_1, ..., and σ_k". We assume that the variables occurring in ϕ include all those occurring in ψ and in the supplementary conditions $\sigma_1, ..., \sigma_k$. The predicate in ϕ is a fluent predicate, while ψ may be fluent or static. However, if ϕ contains variables that do not occur in ψ, then ψ is required to be "upward monotonic", in the sense that no instances of it can become false ($\neg\psi$ does not unify with effect of any operator; this is certainly true if ψ is static). Finally, we require $\sigma_1, ..., \sigma_k$ to be static. The following is an example of this type of constraint in the blocks world stating that the table cannot be on any block: $\forall x, y\ ON(x, y) \Rightarrow NEQ(x, TABLE)$.

Single-valuedness Constraints. An sv-constraint states that the value of a certain predicate argument is unique for any given values of the remaining arguments. An example of an sv-constraint is the following blocks-world constraint stating that any object can be `ON` at most one other object:

$\forall x, y, z.(ON(x, y) \wedge ON(x, z)) \Rightarrow y = z$.

Implicative Constraints + Single-Valuedness Constraints. These invariants are formed by an implicative constraint and a set of sv-constraints that are simultaneously discovered by DISCOPLAN. We distinguish two cases which require different verification conditions: the case of subsumed variables and the case of non-subsumed variables. The blocks-world constraint

`((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y))) (NEQ ?Y TABLE))`

is an example of a combined implicative and sv-constraint for the first case. In general, the implicative constraints we are considering here have as their antecedent a positive literal that contains at least one "starred" variable not occurring in the consequent, and zero or more "unstarred" variables occurring in the consequent. The stars indicate that for all values of the unstarred variables, the antecedent holds for at most one tuple of values of the starred variables.

In the second case we have implications in which both antecedent and consequent contain variables not contained in the other. All such variables are "starred", while the shared variables are unstarred. An example is the following constraint from the `Logistics` domain:

`((IMPLIES (AT ?X ?*Y) (NOT (IN ?X ?*Z))) (OBJECT ?X))`.

This is an *exclusive* state constraint, i.e., it states that no object can simultaneously be `AT` something and `IN` something (and in addition an object can be `AT` no more that one thing, and `IN` no more than one thing).

Antisymmetry Constraints. Antisymmetry constraints are particular implicative constraints of the form

((`IMPLIES` $(P\ t_1\ t_2)$ (`NOT` $(P\ t_2\ t_1)$)) $\sigma_1\ \sigma_2 ... \sigma_n$),

where t_1 and t_2 can be constants or universally quantified variables, and $\sigma_1, ..., \sigma_n$ are supplementary conditions whose variables are a subset of $\{t_1, t_2\}$. An example of an antisymmetry constraint in the blocks world is

$\forall x, y.\ ON(x, y) \Rightarrow \neg ON(y, x)$,

i.e., if one object is on another, then the second is not on the first.

OR and XOR Constraints. OR and XOR-constraints are state constraints of the form

(([`X`]`OR` $\phi\ \psi$) $\sigma_1\ \sigma_2 ... \sigma_n$),

where ϕ and ψ are positive fluent literals, such that non-shared variables are existentially quantified, while shared variables are universally quantified, and where the variables in σ_1, σ_2, ..., σ_n can only be variables shared by ϕ and ψ. An example of an XOR-constraint in the logistics domain is

`((XOR (AT ?X ?Y) (IN ?X ?Z)) (OBJECT ?X))`,

stating that in any reachable state, any object is either `at` some place or `in` something.

Strict Single-Valuedness and n-Valuedness Constraints. This type of invariant is a generalization of sv-constraints. A *nv*-constraint states that a certain predicate can be bound to at most n arguments for any given values of the remaining arguments. A strict nv-constraint states that a certain predicate is bound to exactly n arguments for any given values of the remaining arguments. An example of a strict nv-constraint with $n = 1$ in the blocks world is the invariant stating that any block is on exactly one thing (either another block or the table).

Using "Expanded Operators" to Infer Further Constraints. DISCOPLAN's package includes routines for expanding an operator with a set of given invariants. The operator expansion consists of enriching the operator description with additional preconditions and effects that are entailed by the given invariants. By using expanded operators DISCOPLAN may infer new invariants, which can be used to expand the operators again. This can be iterated until no new constraints are inferred.

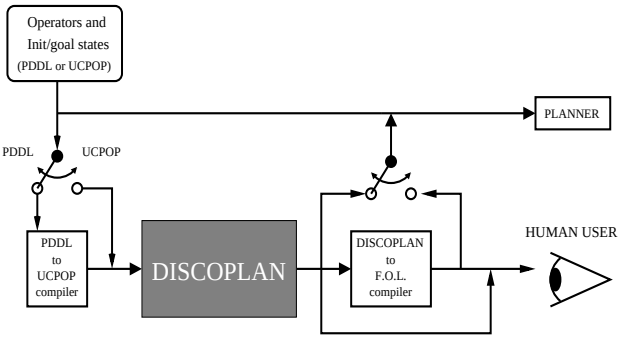


Figure 2: General scheme of DISCOPLAN's input/output

Constraints with Exceptions. Some hypotheses are rejected by DISCOPLAN only because they are not verified against the initial state. For example, consider the blocks world formalization of Figure 1, where we have just one operator in which the parameters are not typed. If we have the simple initial state `((ON A TABLE) (ON C A) (CLEAR C) (ON B TABLE) (CLEAR B))`, DISCOPLAN discovers `(ON ?X ?*Y)`, which then becomes a hypothesis with a strict sv-constraint (`(ON ?X ?Y!1)` in DISCOPLAN format). But `(ON ?X ?Y!1)` is not confirmed because the test against the initial state fails. This is because the object `TABLE` is on nothing in the initial state. In order to deal with these *exceptions*, we have recently weakened the test against the initial state, so that a hypothesis can be verified by restricting the domain of certain variables. In our example `(ON ?X ?Y!1)` can be satisfied in the initial state, provided that `?X` is not instantiated to `TABLE`. Hence, DISCOPLAN weakens the hypothesis by excluding `TABLE` from the domain of `?X`, and derives `((ON ?X ?Y!1) (NOT (MEMBER ?X (TABLE))))`.

These exceptions are computed during the test against the initial state by keeping track of unifiers that assign anomalous tuples of values to the unconstrained variables, i.e., tuples for which strict single-valuedness is violated (e.g., `TABLE`/`?X` in the previous example), and weakening the constraint by excluding these values from the domains of the relevant variables.

This analysis of the initial state is also used to derive additional supplementary conditions rescuing hypotheses that were rejected because a required simultaneous sv-constraint was not satisfied in the initial state (while all the other required verification conditions were satisfied). For example, if in the logistics domain the initial state of a problem contains the facts `(AT ORANGES MIAMI)`, `(AT ORANGES ORLANDO)` and `(OBJECT ORANGES)`, then the invariants

`((IMPLIES (AT ?X ?*Y) (NOT (IN ?X ?*Z))) (OBJECT ?X))`

cannot be inferred. However, DISCOPLAN infers

`((IMPLIES (AT ?X ?*Y) (NOT (IN ?X ?*Z))) (OBJECT ?X)`
`(NOT (MEMBER ?X (ORANGES))))`.[1]

[1]The complete output for these examples can be seen by running DISCOPLAN *on-line* at the web site `http://prometeo.ing.unibs.it/discoplan`.

Interacting with DISCOPLAN on-line

The general input/output scheme of DISCOPLAN is depicted in Figure 2. The input domain and problem descriptions can be specified using the syntax of either UCPOP or PDDL. Since the core functions of DISCOPLAN assume UCPOP descriptions, when the input is specified using PDDL, it is automatically translated into a UCPOP set of operators.

The output of DISCOPLAN can be given as input to either a planner that can exploit this information, or to a domain developer, as an aid to domain specification and debugging. The syntax of the output can be either FOL or the compact format using implicit quantification and "starred" variables as in the previous sections. The compactness of the starred-variable format is due to the fact that it allows an implicative or exclusive constraint to be augmented with simultaneously dicovered sv-constraints merely by starring some variables, rather than adding explicit FOL formulas. The FOL description of the state constraints is obtained by a postprocessing step translating the constraints computed in DISCOPLAN format into FOL.

DISCOPLAN on-line is a version of the system that can be remotely run through any web browser. In particular, from the "test and demo" page of the web site of DISCOPLAN the user can run DISCOPLAN either on a set of predefined domains and problems, or on any other domain and problem that is supplied by the user from her/his local machine (see Figure 3). Before running the system, the user can set some parameters, such as the style of the output, the maximum number of supplementary conditions an invariant can have, the automatic computation of the operator parameter domains using techniques described in [4], etc. Finally, the user can inspect the domain and problem selected.

Related Work and Conclusions

We have sketched how many natural types of state invariants in planning domains with conditional effects can be efficiently inferred, and have described the implemention of our techniques in the DISCOPLAN system. The invariants inferred include predicate domain constraints, relations among static type predicates, implicative constraints, strict and non-strict sv-constraints, combinations of implicative and sv-constraints (where these cannot be inferred in isolation), and OR and XOR constraints. All invariants are found by algorithms that are polynomial-time for any fixed bound on the number of literals in an invariant, and the algorithms can be iterated to find additional invariants after expanding operators using previously found invariants. The outputs can be presented as FOL formulas or in a concise format with implicit universal quantification and "starred" variables indicating single-valuedness. The automatically derived invariants have been shown to radically boost the performance of SAT planners, and are also potentially useful for other planning styles, and as a help in domain analysis and debugging.

Other approaches for the automatic inference of state invariants have been proposed including [10, 9, 3, 13, 14, 12], but to the best of our knowledge the only other implemented system that is available is Fox and Long's TIM. A major difference between the DISCOPLAN and these approaches is that DISCOPLAN can process domains specified using a more expressive planning language. In particular, TIM does not handle operators with conditional effects and negated preconditions. Moreover, DISCOPLAN infers some types of constraints that are not inferred by TIM, such as antisymmetry constraints, XOR-constraints and some implicative constraints involding variable binding constraints or predicates without parameters.[2] On the other hand, some of TIM's "state membership invariants" and "uniqueness invariants" are not inferred by the currently implemented version of DISCOPLAN.

Figure 3: Test and Demo page of DISCOPLAN on-line

It remains unclear how important the "omissions" in each system, relative to the other, are for planning and domain analysis purposes. In any case a reasonable strategy at this time, for builders of planning systems that can benefit from state invariants, would be to combine the invariants found by TIM and DISCOPLAN.

We have developed some further algorithms for inferring invariants, beyond those implemented in DISCOPLAN. The most general of these is an algorithm for inferring n-ary disjunctions of fluent literals, together with sv-constraints and static supplementary conditions, for n not limited to 2 (as at present). This algorithm is a candidate for future implementation.

References

[1] A. Blum and M.L. Furst. Fast planning through planning graph analysis. In *Proc. of IJCAI-95*, pp. 1636–1642. 1995.

[2] M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proc. of IJCAI-97*, pp. 1169–1176. 1997.

[3] M. Fox and D. Long. The automatic inference of state invariants in TIM. *JAIR*, 9:367–421, 1998.

[4] A. Gerevini and L. Schubert. Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. *JAIR*, 5:95–137, Sept. 1996.

[5] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proc. of AAAI-98*, pp. 905–912. 1998.

[6] A. Gerevini and L. Schubert. Inferring state constraints in DISCOPLAN: Some new results. In *Proc. of AAAI-00*, pp. 761–767. 2000.

[7] M. Ghallab, A. Howe, G. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – planning domain definition language. Available at `http://cs-www.cs.yale.edu/homes/dvm/`.

[8] H.A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of AAAI-96*, 1996.

[9] G. Kelleher. Determining general consequences of sets of actions. Technical Report TR CMS.14.96, Liverpool Moores University, 1996.

[10] J. Kelleher and A. Cohn. Automatically synthesising domain constraints from operator descriptions. In *Proc. of ECAI-92*, pp. 653–655. 1992.

[11] J.S. Penberthy and D.S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. of KR'92*, pp. 103–114. 1992.

[12] U. Scholz. Extracting state constraints from PDDL-like planning domains. In *Working Notes of the AIPS00 Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, pp. 43–48. 2000.

[13] J. Rintanen. A planning algorithm not based on directional search. In *Proc. of KR'98*, pp. 617–624. 1998.

[14] J. Rintanen. An iterative algorithm for synthesizing invariants. In *Proc. of AAAI-00*, pp. 806–811. 2000.

[2] Examples of these constraints are: `((IMPLIES (ON ?X ?Y) (NEQ ?X ?Y)))` in the blocks works, and `((IMPLIES (HASBANANAS) (HASKNIFE))` in the Monkey domain.

CODA: Coordinating Human Planners

Karen L. Myers, Peter A. Jarvis, and Thomas J. Lee

AI Center, SRI International, Menlo Park, CA USA *

Abstract. Effective coordination of distributed human planners requires timely communication of relevant information to ensure the overall coherence of activities and the compatibility of assumptions. The CODA system provides targeted information dissemination among distributed human planners as a way of improving coordination. Within CODA, each planner declares interest in different types of plan changes that could impact his or her local plan development. As individuals develop plans using a plan authoring tool, their activities are monitored; changes that match declared interests trigger automatic notification of appropriate planners. In this way, distributed planners can receive focused, real-time updates of plan changes that are relevant to their local planning efforts.

1 Introduction

The scope and complexity of large-scale planning tasks often necessitates cooperation among multiple planners with differing areas of expertise, each of whom contributes portions of the overall plan. These planners may be distributed both geographically and temporally, thus further complicating coordination.

As a concrete illustration, consider special operations forces (SOF) mission planning for the military (the motivating domain for our work). SOF planning involves numerous people working on separate but interconnected facets (e.g., strategy, logistics, medical, intel) of an overall plan. The SOF planning process is time constrained, concurrent, and iterative. Individual planners construct subplans based on their expectations for the operating environment and requirements. As the overall plan develops, expectations evolve and modifications must be made. Currently, such changes are communicated informally by word of mouth, or transmitted in batch mode at regularly scheduled coordination sessions. This approach can lead to omissions and delays that reduce the effectiveness of the planning process and the quality of the resulting plans.

The SOF planning domain lies well beyond the range of current automated planning technologies. Moreover, fully automated approaches are unlikely ever to succeed because (a) planning for this domain involves a huge strategic component that is extremely difficult to model, and (b) successive planning tasks (e.g., disaster relief, counter-terrorism) tend to be unique, making it difficult to formulate reusable background knowledge with adequate coverage.

Techniques from the AI planning community can still contribute to complex domains of this type. In particular, *plan authoring* tools can improve the plan

* This work was supported by DARPA Contracts F30602-97-C-0067 and F30602-00-C-0058, under the supervision of Air Force Research Laboratory – Rome.

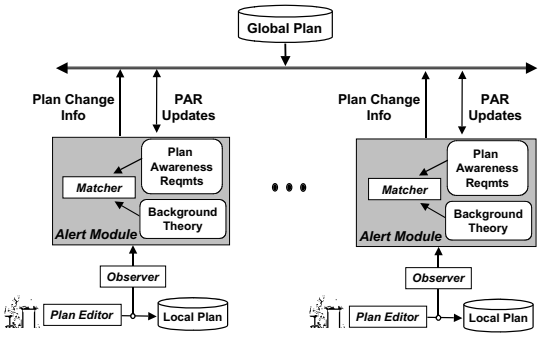


Fig. 1. CODA Architecture

development process [2, 3]. Plan authoring tools provide a structured set of plan editing operations that support users in developing large-scale plans, yielding principled representations of plans with well-defined semantics. Their main role is to augment rather than replace human planning skills, but they may provide limited automated capabilities. Planning aids can be defined that reason over the plan structures produced by these tools, including capabilities for supporting multiplanner coordination.

The CODA system (*Coordination of Distributed Activities*) provides automated support for focused information sharing during collaborative plan development by a team of humans using plan authoring tools [5]. CODA is designed for applications where distributed human planners are assigned responsibility for developing subportions of a global plan. These subplans are expected to have a moderate degree of coupling due to the need to reflect coherent strategy, to coordinate actions, and to share limited resources. Within CODA, each planner declares the kinds of plan changes that are of interest to him or her; we call these declarations *plan awareness requirements* (or *PARs*). As users develop plans with a plan authoring tool, their activities are monitored. Changes that match detected PARs are forwarded automatically to the person who declared interest in them. In this way, distributed planners can receive focused, real-time updates of plan changes that are relevant to their local planning efforts.

2 CODA Architecture

Figure 1 presents the architecture of CODA. Within the context of a global plan, individuals work independently to produce local plans for their assigned tasks. Plans are developed using a structured *plan editor*, which supports a broad range of plan manipulation capabilities. User interactions with the plan editor are tracked by an *observer* module, which maintains a history of all editing operations. As events are logged, a semantically grounded representation of the local plan is built.

The *matcher* provides the main inferential capability within CODA, being responsible for linking observed plan changes to declared PARs. The matching

process may involve reasoning with a *background theory*, whose role is to bridge the gap between low-level plan edits and the high-level languages used to define PARs. When matches are detected, notification is sent to the local planner who registered the matched PAR.

CODA could be linked to a variety of manual and automated planning tools. Currently, it is connected to the SOFTools Temporal Planner – a plan authoring tool that supports graphical editing of SOF mission plans [2]. CODA's event monitoring for the Temporal Planner covers most of the available editing operations, including creation, modification and deletion of objects, modification of object attributes, temporal shifting of activities, and resource assignment.

CODA supports two modes for controlling PAR matching. In *real-time* mode, PARs are checked after every plan edit, thus providing immediate notification to planners of relevant changes. Real-time notification is suitable for the endstages of planning or execution, when plans are mostly stable and changes could be significant. For earlier stages of plan development, when changes would be frequent and wide ranging, CODA provides a *batch* mode of matching. Users can invoke batch mode to summarize PAR matches for a sequence of plan modifications relative to a designated checkpoint plan.

3 Plan Awareness Requirements

The PAR representation language builds on a general-purpose query language for the CODA plan ontology. It consists of a typed first-order language that builds in a model of frame representation systems as well as equality, term constructors for lists and intensionally defined sets, and quantification with respect to an enumerable type. CODA supports two types of PAR: *plan-state* and *transition*.

Plan-state PARs describe conditions of a plan and are modeled in terms of a formula in the plan query language. For example: *There is an arrival to staging base Gold scheduled for after 8 PM.* Matching of a plan-state PAR occurs when a modification results in a plan that satisfies the associated plan query.

Plan transition PARs describe changes between two plan states. We distinguish several categories, based on the nature of the underlying plan changes:

- *Instance Creation PARs* are used to declare interest in the addition of an object to a plan that satisfies stated conditions. For example: *Addition of decision points related to weather calls.*
- *Instance Deletion PARs* are used to declare interest in the removal of an object from a plan that satisfies stated conditions. For example: *Elimination of a landing zone south of the embassy.*
- *Instance Modification PARs* are used to declare interest in the modification of an object that satisfies stated conditions. For example: *Changes to movements by the 4th-platoon.*
- *Attribute Modification PARs* specialize Instance Modification PARs to changes to a specific attribute of a plan object, possibly satisfying stated change conditions. For example: *Delays of $>$ 1 hour in time to secure the Church.*
- *Aggregate Modification PARs* can be used to declare interest in changes to an intensionally defined collection of objects. The change may be to membership

in the collection, or to some *aggregation value* defined over the collection. As an example: *Decrease of > 2 in the number of fire-support aircraft.*

CODA includes an interactive *PAR authoring tool* that helps users define the plan changes in which they are interested. This tool builds on the Adaptive Forms technology [1], a grammar-based framework that supports the specification of structured data through a form-filling interface that adapts in response to user inputs. With this tool, users create PARs by filling in forms using an English-like syntax; as users incrementally specify PARs, remaining options adjust in accord with constraints of the underlying grammar. An internal compiler transforms the completed forms into the formal PAR structures required by CODA's matcher.

When designing user input tools, the competing requirements of expressivity and ease of use must be balanced. To address this issue, CODA's PAR authoring tool provides two sets of forms. First, a set of *general forms* provides the full expressive power of the PAR language. While powerful, these forms require more effort to complete; in addition, people unaccustomed to formal languages require training to use them effectively. Second, the tool includes specializations of the general forms that capture commonly used *idioms* within the SOF planning domain. The specialized forms build in values that users would have to specify in the general case, thus simplifying the specification process.

4 Conclusions

CODA provides a practical solution to the problem of coordinating distributed human planners. By having human planners explicitly declare those aspects of the overall planning process that interest them, CODA can provide timely and focused distribution of information that will expedite and improve the quality of coordinated problem solving. The use of a rich, AI-based representation for plans and planning operations provides the key to this technology.

The AI Planning community has developed several systems that share information to coordinate multiple automated planners, using techniques such as constraint propagation [4] and relevance reasoning [6] that analyze the causal structure of local plans. These approaches do not transfer to settings where humans author plans because complete causal structures, while a by-product of automated planning methods, will not be available.

References

[1] M. Frank and P. Szekely. Adaptive Forms: An interaction paradigm for entering structured data. *Proc. of the ACM Intl. Conf. on Intelligent User Interfaces*, 1998.

[2] GTE. *SOFTools User Manual*, January 2000.

[3] C. Knoblock, S. Minton, J. Ambite, M. Muslea, J. Oh, and M. Frank. Mixed-initiative, multi-source information assistants. *Intl. World Wide Web Conf.,* 2001.

[4] A. L. Lansky. Localized planning with action-based constraints. *Artificial Intelligence*, 98(1-2), 1998.

[5] K. L. Myers, P. A. Jarvis, and T. J. Lee. CODA: Coordinating distributed human planners. Technical Report, AI Center, SRI International, Menlo Park, CA, 2001.

[6] M. J. Wolverton and M. desJardins. Controlling communication in distributed planning using irrelevance reasoning. In *Proc. of AAAI-98*, 1998.

An Integrated Planning and Scheduling Prototype for Automated Mars Rover Command Generation

Rob Sherwood, Andrew Mishkin, Steve Chien, Tara Estlin,
Paul Backes, Brian Cooper, Gregg Rabideau, Barbara Engelhardt

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Dr., Pasadena, CA 91109
firstname.lastname@jpl.nasa.gov

Abstract. With the arrival of the Pathfinder spacecraft in 1997, NASA began a series of missions to explore the surface of Mars with robotic vehicles. The mission was a success in terms of delivering a rover to the surface, but illustrated the need for greater autonomy on future surface missions. The planning process for this mission was manual, and very time constrained since it depended upon data from the current day to plan the next day. This labor-intensive process was not sustainable on a daily basis for even the simple Sojourner rover for the two-month mission. Future rovers will travel longer distances, visit multiple sites each day, contain several instruments, and have mission duration of a year or more. Manual planning with so many operational constraints and goals will be unmanageable. This paper discusses a proof-of-concept prototype for ground-based automatic generation of rover command sequences from high-level goals using AI-based planning software.

1 Demonstration

We will demonstrate a ground based automated planning prototype for a multi-instrument Mars rover using the ASPEN planner (Chien, et al., 2000). With this software, new goals can be added to the existing plan, resulting in conflicts that will be solved using an iterative repair algorithm. The end result will be a valid sequence of commands for execution on a rover.

2 Introduction

Over the next 10 years, NASA will be sending a series of rovers to explore the surface of Mars. The rover planning process uses specialized tools for path planning and instrument planning, but the actual activity planning and scheduling is a manual process (Mishkin, et al., 1998). We are using AI planning/scheduling technology to automatically generate valid rover command sequences from goals specified by the specialized tools. This system encodes rover design knowledge and uses search and reasoning techniques to automatically generate low-level command sequences while respecting rover operability constraints, science and engineering preferences, environmental predictions, and also adhering to hard temporal constraints.

3 ASPEN Planning System

In ASPEN, the main algorithm for automated planning and scheduling is based on a technique called *iterative repair* (Rabideau, et al., 1999, Zweben et al., 1994). During iterative repair, the conflicts in the schedule are addressed one at a time until conflicts no longer exist, or a user-defined time limit has been exceeded. A conflict occurs when a resource requirement, parameter dependency or temporal constraint is

not satisfied. Conflicts can be repaired by means of several predefined methods. The repair methods are: moving an activity, adding a new instance of an activity, deleting an activity, decomposing an activity into subgoals, abstracting an activity, making a resource reservation on an activity, canceling a reservation, connecting a temporal constraint, disconnecting a constraint, and changing a parameter value. The repair algorithm may use any of these methods in an attempt to resolve a conflict. How the algorithm performs is largely dependent on the type of conflict being resolved and the activities, states, and resources involved in the conflict.

4 Rover Motion Planning

ASPEN is able to reason about simple resource and state constraints. ASPEN also has the ability to use simple external functions to calculate parameters for resource usage. Many rover constraints are too complex to reason about in a generalized planning system, or use simple parameter functions to solve. For these, an external program must be used to reason about these constraints. ASPEN can interface with other domain-specific programs (or special purpose algorithms) using input files, library calls, a socket interface, or software interfaces. Motion planning is a good example of a complex rover constraint requiring a specialized tool.

JPL uses a tool called Rover Control Workstation (RCW) for the motion-planning problem (Cooper, 1998). RCW provides a unique interface consisting of a mosaic of stereo windows displaying the panorama of Mars using camera images from both a lander and a rover. The operations team uses the RCW to make decisions about where to safely send the rover and what to do when reaching the goal. RCW calculates the maximum safe tilt angles for the rover traverse goals input by the user. RCW also calculates the parameters for the rover motion commands. The RCW software outputs a set of goals that cannot be changed in ASPEN.

5 Mixed-Initiative Rover Planning

While the goal of this work is an integrated fully automated planning system for generating a rover sequence of commands, the human operator is required to be part of the planning process. There is not enough CPU capability onboard current flight rovers to run autonomy software such as path planning or generalized planning. The JPL developed Web Interface for Telescience (WITS) science-planning tool (Backes, et al., 1998) and the RCW motion-planning tool each require human interaction. These tools allow the user to select rover destinations and science targets in three dimensions using surface imagery. The WITS tool does not actually enforce an order of the goals, but instead relies on ASPEN to build the plan, schedule, and check the resource usage.

Combining these tools with ASPEN creates a “mixed-initiative” end-to-end planning system. The ASPEN operator starts with a set of goals from WITS and RCW, but can then modify the schedule within ASPEN by inserting new goals, changing existing activities, or deleting activities. The schedule is then generated using a forward dispatch algorithm followed by an iterative repair algorithm to fix any conflicts. The repair actions available for each activity are defined within the model for that activity. If the rover resources are over-constrained or under utilized, the user may decide to modify the schedule to optimize the rover resource usage, then re-run the iterative repair algorithm. Several iterations can be performed using ASPEN, WITS, or RCW to modify the goals. This capability allows the rover operations team to try several different scenarios before deciding on the best course of action. The result of this mixed-initiative optimization strategy is a plan with increased science

opportunities. Because ASPEN is autonomously checking flight rules and resource constraints, the plan should also be safer than a manually generated plan.

We are also investigating how the user should be interacting with each of the tools involved in building a schedule. The science and engineering users are used to interacting with WITS and RCW, but not with ASPEN. Yet WITS and RCW do not show resource information and activity ordering. Currently the system requires the user to utilize the ASPEN GUI for resource and activity information. In the future, this information could be added directly to the WITS GUI.

6 Difficulties in Modeling Rover Constraints

There are several aspects of modeling the Mars rover domain has proven to be very difficult. The power system is a good example. The rovers planned for 2003 contain solar arrays and rechargeable batteries. During the daytime, the power for rover operations is produced using the solar arrays. If the total power drain from operating the rover exceeds the available power from the solar arrays, the batteries must be drawn upon. Because the battery drain is context dependent, the planner needs to understand all the influences and be able to repair conflicts using this knowledge. Additionally, computing the energy taken from a battery is a function of the battery parameters such as temperature, current, voltage, etc. Representing this in a planning model is very difficult.

To solve the power-modeling problem, we initially used a parameter dependency function to calculate the amount of solar power and battery power as a function of the activity duration, available solar array power, available battery power, and power required by the activity. This technique will only work if there are no overlapping power activities because the calculated solar array and battery usage are based on the amount available at the beginning of the activity. In the ASPEN representation, resource use is assumed to be constant over the duration of the activity. In the same manner, we can only request the existing value of a resource at the start of the activity and we must assume that the existing resource profile remains constant until the end of the activity. In the case of overlapping activities that consume power, the first of the two activities would calculate the required power based on the available power at the start time of the first activity. The power available would change during the activity due to the overlap of the second activity.

Another difficulty with modeling the depletable resources in planning systems is the usage profile. Some examples in the spacecraft and rover domains include the memory buffer resource, battery, and fuel. If an activity that uses the memory buffer resource has duration of several minutes, ASPEN will change the value of the resource timeline at the beginning of the activity. In this case, the entire amount of memory buffer resource used by the activity is unavailable for the entire activity. In the example, the memory resource is set to them maximum value at the start of the timeline. This is the equivalent of consuming an entire tank of gas in a car at the beginning of a trip rather than using the gas gradually over the course of the trip. Likely the actual resource usage is linear over the duration of the activity. For long activities, the depletable resource value near the beginning of the activity can be very inaccurate. One workaround for this problem is to split the activity up into several subactivities, each using an equal fraction of the resource. This solution has several problems. First, it increases scheduling complexity by adding multiple activities into the activity database. Second, it creates the problem of trying to determine how many subactivities is enough to accurately model the resource usage. Third, it's non-intuitive for the user to see multiple subactivities that don't represent actual events. The ideal

method for modeling resource usage is to use a generalized timeline. Generalized timelines allow modelers to provide a set of functions to describe the depletable resource timeline and its constraints. The generic scheduler can then accurately reason about the described timelines. The example given contains a linear depletable timeline, but any other function could have been modeled as well.

7 Conclusions

Planning and reasoning about complex rover resources is a difficult task to automate. The rover planning process involves interfacing with other specialized planning tools to create a mixed-initiative end-to-end planning system.

Current approaches to rover-sequence generation and validation are largely manual, resulting in a labor and knowledge intensive process. This is an inefficient use of scarce science-investigator and key engineering-staff resources. Automation as targeted by this tool will automatically generate a constraint and flight rule checked, time ordered list of commands and provide resource analysis options to enable users to perform more informative and fast trade-off analyses.

8 Acknowledgement

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

P. Backes, K. Tso, and G. Tharp. "Mars Pathfinder mission Internet-based operations using WITS. Proceedings IEEE International Conference on Robotics and Automation," pages 284-291, Leuven, Belgium, May 1998.

B. Cooper, “Driving On The Surface Of Mars Using The Rover Control Workstation,” SpaceOps 98, Japan, 1998.

S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, D. Tran , "ASPEN - Automating Space Mission Operations using Automated Planning and Scheduling," SpaceOps, Toulouse, France, June 2000.

A. Mishkin, "Field Testing on Mars: Experience Operating the Pathfinder Microrover at Ares Vallis," presentation at Field Robotics: Theory and Practice workshop, May 16 1998, at the 1998 IEEE International Conference on Robotics and Automation, Leuven, Belgium.

G. Rabideau, R. Knight, S. Chien, A. Fukunaga, A. Govindjee, "Iterative Repair Planning for Spacecraft Operations in the ASPEN System," International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS), Noordwijk, The Netherlands, June 1999.

Zweben, M., Daun, B., Davis, E., and Deale, M., “Scheduling and Rescheduling with Iterative Repair,” Intelligent Scheduling, Morgan Kaufmann, San Francisco, 1994, pp. 241-256.

GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning

R. M. Simpson, T. L. McCluskey, W. Zhao,†
R. S. Aylett, C. Doniat ‡

Department of Computing Science, University of Huddersfield,Queens Gate,Huddersfield, HD1 3DH, UK †
Centre for Virtual Environments, University of Salford, Salford, M5 4WT, UK ‡
email: r.m.simpson, t.l.mccluskey, w.zhao @hud.ac.uk
email: R.S.Aylett, c.doniat @salford.ac.uk

Abstract We describe a Graphical Interface for Planning with Objects called GIPO that has been built to investigate and support the knowledge engineering process in the building of applied AI planning systems. GIPO embodies an object centred approach to planning domain modelling. There are two reasons for providing knowledge engineering support for AI planning: (i) to apply a planning system to a new domain to test the planning system itself (ii) to tackle the end-user problem for the engineer who might be a domain expert but need not necessarily have a specialist knowledge of AI planning. Our research is primarily aimed at developing a method and tools to meet the requirements of the latter case (ii), although the benefits can also be enjoyed by planning experts.

1. Introduction

Planform [1] is a UK EPSRC grant funded research project in which we are developing an open platform for the systematic acquisition of planning domain models, and tools to combine these models with planners to create efficient planning applications. Part of the work involves the development of an knowledge acquisition method where knowledge is captured by describing changes that the objects in the domain undergo as the result of the application of operators. The method requires that we structure the domain definition around types of objects, the states that these objects may inhabit, and the possible transitions from state to state that the objects may undergo as a result of the application of planning operators. The content of this definition provides the basis for much of the validation and cross-checking that the tool is capable of performing and allows the domain developer to approach the task of defining operators in a structured and well-supported manner. The additional support provides the possibility of opening up domain definition to modellers who do not need to be as skilled in AI planning technology as has traditionally been the case.

In brief, *GIPO* provides (a) a graphical means of defining a planning domain model (b) a range of validation tools to perform syntactic and semantic checks of emerging domain models (c) dynamic tools to allow the modeller to verify that the domain specification can support known plans within the domain (d) tools to import and export domain definitions to the literal-based PDDL format for typed strips domains, with or without conditional effects (e) an interface that allows for the integration of third party planning algorithms to be run and animated from within the tools environment.

GIPO is designed on the assumption that the knowledge engineer will be trying to build descriptions of new domains using a method which imposes a loose sequence on the sub-tasks to be undertaken to develop an initial model. Once an initial rough model has been constructed, development may proceed in a more iterative and experimental manner. A key design goal in building the supporting GUI tool has been to allow the creation of a specification with the tool taking care of the detail of the syntax of the underlying specification. Use of the tool will never result in a syntactically ill-formed specification.

2. Domain Acquisition

The process of domain model development and the model's ontology on which this is based is detailed in the literature (see GIPO home page [2], which also contains a more detailed version of this paper). Here we sketch the main steps of the knowledge acquisition process, describing how the tool supports this process. We outline two important steps of the knowledge acquisition process - acquiring domain structure and acquiring domain actions.

The process starts with the identification of the kinds of objects that characterise the domain. The method requires that distinct collections of objects, which we call *sorts*, can be organised into a hierarchy. A visual tree editor is used to construct a sort tree and the relations and attributes that characterise the objects of each sort. The key step in the object centred modelling process is to characterise each valid state of objects of the sorts that are subject to change during the planning process by defining their relations and attributes. We refer to sorts subject to such change as *dynamic* where as the sorts where the objects remain unchanged are *static*. A description of a state of an object we call a *substate definition*. Under classical assumptions each member object of a sort may be in only one such substate at any time, and that during plan execution the object goes through *transitions* which change its state from one such substate to another.

In parallel with the specification of substates the modeller can now assemble planning operators. The operator editor forms the heart of GIPO. This editor relies on the notion that operators and methods generally cause objects to change from one substate to another (called *object transitions*). Whereas substate definition captures domain structure, operator definition captures domain behaviour.

For each object changed by the application of an operator there will be a transition defining the set of substates the object may be in prior to the application of the operator and the definition of the precise substate the object will be in as a result of applying the operator. We enable the composition of operators by the domain modeller building a simple graph of the operator by selecting the elements of the transitions from an available list of the predefined substates the object/sort is capable of being in. Consider the remove wheel operator taken from a *tyre change* domain illustrated in figure 1. The rectangles describe states or generalisations on states of objects of identified sorts where the pair of rectangles in the same row represent the *transition* that the referenced object will make as a result of applying the operator, named in the *oval box*. Here there are two objects changed by the operation, the wheel itself and the hub that the wheel was attached to. The hub in the example moves from the state where it is jacked up and free to being jacked up and bare as a result of removing the wheel.

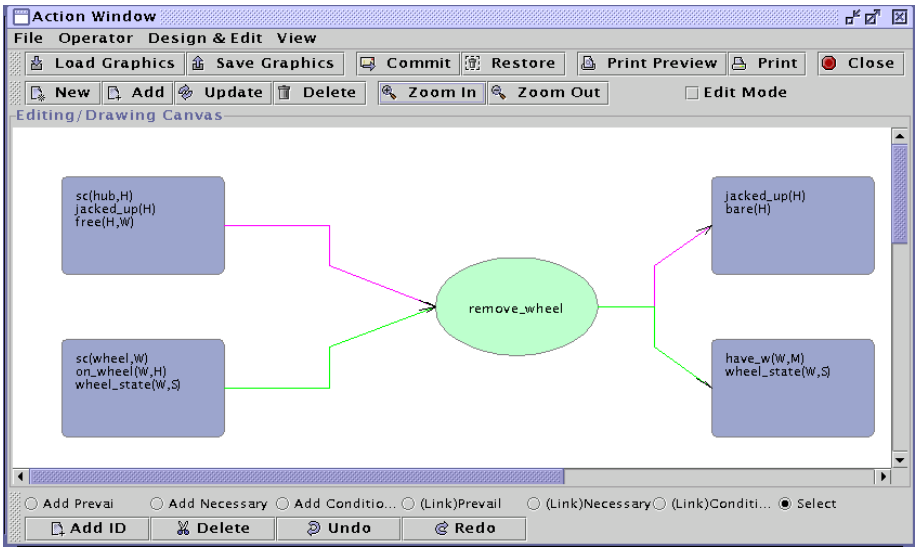


Fig. 1. The Operator Definition Editor

3. Domain Analysis

During domain model acquisition numerous *local verification checks* are applied to ensure consistency. Once an initial domain model has been defined as described above, the domain modeller can run *global verification tools* to further check the validity of the specification. Tools which we have developed include goal ordering generators, a random tasks generator, and a "reachability" analysis tool. The latter tool examines substates that are defined for a sort and indexes them against the operators that use them either as consumers or producers. This may reveal to the domain modeller that contrary to expectation some substates cannot be produced and hence could only ever be used in the initial state of an object, or that some cannot be consumed and hence either are only useful in the development of objects of other sorts, or are of the kind of specified only in a goal condition.

In addition to static analysis of the specification the domain modeller can dynamically check a domain against a set of problems either by using the manual *stepper* (shown in figure 2) or by running a selected planning algorithm against defined test problem cases. With the stepper, the engineer chooses actions to apply in the current state to generate the consequent state and proceeds in this manner to verify that the domain and operator definitions do support known plans for given problems within the domain. In the example shown in figure 2, again drawn from the *tyre change* domain, each column of circles represents the state of objects at one time instance. The linked oval is the operator applied to the states with the links tracing the objects changing and participating in the application of the operator. The inset dialog box shows an operator *fetch_tool* in the process of the user choosing instantiations for the parameters. The panes at the sides show the task being attempted and a list of available operators in the domain.

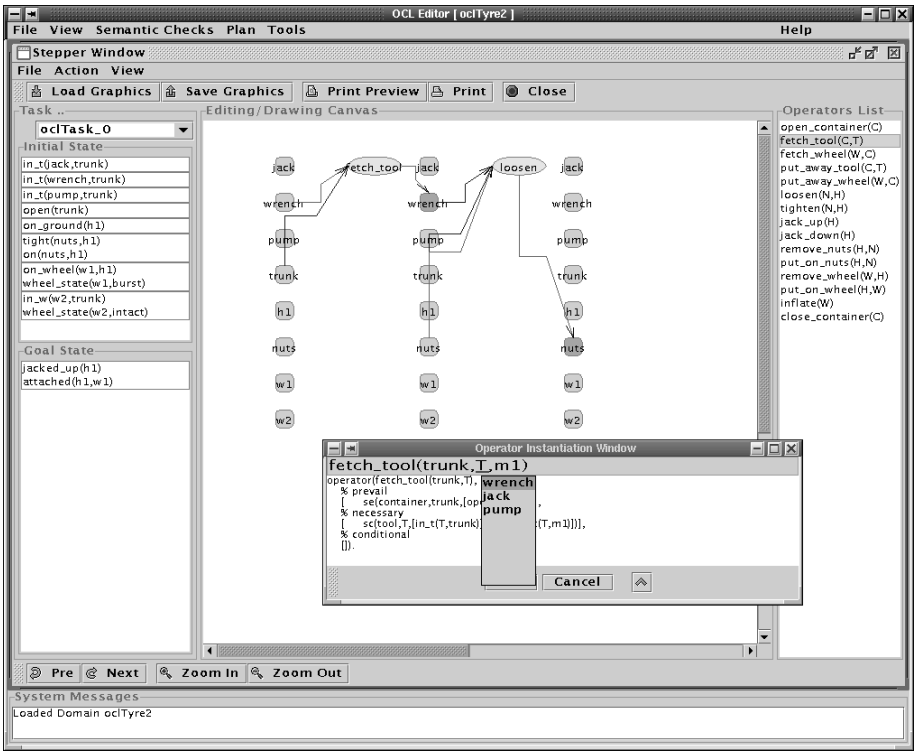


Fig. 2. The Plan Stepper

The *animator* allows integrated planners to run against defined problems and graphically displays the transitions made to objects as the plan unfolds. The animator is structurally very similar to the stepper except that the operator choice is determined by the results output by the planner.

4. Future Work

Although the object centred method lifts domain acquisition to a conceptual level, the details of specifying substate definitions and transitions are still too theoretical for an unskilled user. We aim in the future to incorporate more inferencing mechanisms to aid the unskilled user in this task. We are also developing methods to assist the domain modeller to extract structured knowledge from informal textual descriptions of the domain and to assist the modeller to create new models by providing a library of previous domain models.

References

[1] http://scom.hud.ac.uk/planform
[2] http://scom.hud.ac.uk/planform/gipo

ECP-01 Sponsors

Artificial Intelligence Journal

PLANET 2
The Network of Excellence in AI Planning

Ministerio de Ciencia y Tecnología (MCYT)

APSOLVE
A BTexaCT Technologies Company

AI*IA
Italian Association for Artificial Intelligence

IP-CNR
Institute of Psychology of the Italian National Research Council

Universidad Carlos III de Madrid

ScALAB
Systems, Complex and Adaptive Laboratory

PST
Planning and Scheduling Team at IP-CNR