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.