Planner R: A variant of STRIPS

Fangzhen Lin (flin@cs.ust.hk)
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

Our planner R is a variant of the original STRIPS by Fikes and Nilsson. Like STRIPS, it is a linear planner that solves a goal one at a time. But it also differs from STRIPS in some important aspects such as the way the planner decides to achieve a goal that cannot be made true immediately by an executable action. It also has some control mechanism for the user to provide domain specific information to control it. In this paper, we shall describe the algorithm used by R and present some experimental results from AIPS-2000 planning competition in which R was competed.

1 Introduction

The original STRIPS by Fikes and Nilsson [1971] (called forward-chaining, recursive STRIPS in [Nilsson, 1998]) used a simple algorithm for planning. Briefly speaking, given a conjunction of simple goals, it selects one of them to work on. If there is an action that is executable and will make this simple goal true once executed, then it does this action, updates the global database, and then recursively tries to achieve the other simple goals in the new situation. Otherwise, it finds an action that have this simple goal as one of its effects, and recursively tries to achieve the preconditions of this action. This process continues until the original goal is achieved.

This algorithm, while inspired much of the later work in planning and is described in almost all introductory AI textbooks, is widely perceived to be too simple to be effective. However, as we shall argue in this paper, it can be made effective, at least for some of the well-known benchmark domains, by using a better subgoalling strategy and a loop checking mechanism. Specifically we shall describe our planner R that uses a planning algorithm that is similar to the one used in STRIPS. This planner was competed in the recent AIPS-2000 planning competition[1] in both the fully automated and hand-tailored tracks, and was selected by the organizers as one of distinguished planners.

[1] See http://www.cs.toronto.edu/aips2000.

2 Logical Preliminaries

We shall assume a first-order language. By a *simple goal* we mean an atomic formula, and by a *goal* we mean a conjunction of simple goals. In the following, we shall equate a finite set of formulas with its conjunction. We shall assume here that actions are context-free, i.e. STRIPS actions. We are making this assumption not because that our algorithm works only for these actions, but in order to simplify our description of the algorithm and to make comparisons with STRIPS easier.

There are many ways to represent (context-free) actions. Here we use the common STRIPS notation to represent an action as a triple: a precondition list, a delete list, and an add list, where the lists are finite sets of simple goals whose variables are parameters of the action. As shown in [Lin and Reiter, 1997], a triple like this can be considered to be shorthand for a situation calculus theory of a special form.

Given a set of actions, a planning problem is then given by an initial situation description, which is a finite set I of atomic sentences, and a ground goal G. A sequence α of ground actions is a plan for such a problem if the *progression* of the initial situation description I over α contains G, where the progression of a finite set S of atomic sentences over a finite sequence β of ground actions is defined inductively as follows:

- if β is empty, then the progression is S itself;
- inductively, let the head of β be A, and the tail β';
  - if A is executable in S, i.e. the precondition list of A is contained in S, then the progression is the progression of
  $$\{p \mid p \in S, p \notin Del(A)\} \cup Add(A)$$
  over β', where $Del(A)$ is the delete list of A and $Add(A)$ is the add list of A;
  - if A is not executable in S, then the progression is undefined.

3 The planning algorithm - an informal description

Informally, this is how the algorithm used in R works: given a conjunction of simple goals, it selects one of them to work on. If there is an action that is executable, and will make this simple goal true once executed, then it does this action and then tries to achieve the other simple goals in the new situation. Otherwise, it computes a new conjunctive goal based on the actions that have this simple goal as one of their effects, and recursively tries to achieve the new goal. This process continues until the original goal is achieved. As one can see, the basic idea is similar to that in STRIPS. However, there are some important differences:

- STRIPS maintains a global database which on starting up is the initial state description. During the planning process, it updates this global database so that it is always the description about the current situation. This means that once STRIPS finds an action to apply, it commits to it, and no backtracking can undo that. In comparison, our algorithm does not modify the initial state description. Instead, it maintains a global list Γ that represents the sequence of actions that the planner has found so far at the moment. But on backtracking, as for example on reaching a deadend when pursuing a simple goal, some of the actions in Γ may be backtracked. This makes our planner more flexible. On the other hand, this also means that checking if a fluent is true in the current situation is costlier in our algorithm than in STRIPS. Interestingly, RSTRIPS, a variant of STRIPS given in [Nilsson, 1980], does not change the initial situation description either, and relies on regression to compute the truth value of a fluent in any future situations.
- When attempting to achieve a simple goal, our algorithm keeps track of the sequence of all the simple goals that the planner has subgoaled to and makes sure that no cycles are involved. That is, it does not do something like "to achieve g, try to achieve g first." This seems to be a crucial factor in making our algorithm effective on some of domains at the competition.
- When attempting to achieve a simple goal g that cannot be made true immediately by an executable action, STRIPS selects an action that can make g true, and subgoals it to the preconditions of this action. In general, some of these conditions may contain variables, and these variables are eventually grounded by matching them with the current state description. For our planner, however, the user can decide how to achieve a simple goal using domain specific information. We will describe this in more detail later in the section on controlling the planner. If there are no user provided information, the planner uses the following strategy:
  1. compute first the common ground preconditions of all ground actions that can make g true. If any of them is not yet true in the current situation, then the conjunction of these conditions is chosen to be the new goal. If all of them are true already, then it performs the next step.
  2. selects an action that can make g true; consider first the conjunction of all *ground* preconditions of this action. If this conjunction is not yet true in the current situation, then choose it as the new goal. If it is already true, then let the new goal be the conjunction of all the preconditions of this action with all the variables instantiated by some constants.

  Notice that in all cases, the new goal is always grounded.

In the next section, we describe our planning algorithm more precisely. Readers who do not wish to go into technical details can jump to section 5 for an example in the blocks world that illustrates the planning algorithm.

4 The planning algorithm

The following is a more detailed description of the algorithm used by our system. It is implemented by two mutually recursive functions FIND-PLAN(G) and FIND-PLAN-SG(g). They share a common initial situation description, and two global stacks: Γ, used to store the sequence of actions the planner is currently committing to, and Π, used to store the sequence of simple goals that the planner is currently working on. Both of the stacks are empty initially. The recursive function FIND-PLAN(G), if terminates, will return a plan for the conjunctive goal G. It calls FIND-PLAN-SG(g) for finding a sequence of actions for the simple goal g.

Function FIND-PLAN(G).

1. **repeat**
2. let g be a conjunct of G that has not been achieved by Γ. The default is to select the first such one in G. This is a backtracking point.
3. **if** g is already in Π, **then** fail **else** push g into Π.
4. FIND-PLAN-SG(g). This recursive function tries to find a plan that will achieve the simple goal g. It will normally modify the global stack Γ. In fact, after this function returns, Γ will be a plan for the simple goal g. It will also modify Π but will always restore it to its initial state which has g at the top.
5. Pop g from Π.
6. FIND-PLAN(G).
7. **until** G is satisfied in the state resulted from performing Γ in the initial situation.

Function FIND-PLAN-SG**(g)**

1. **repeat**
2. SOLVE(g). This function is at the heart of the planner. It decides how to achieve the simple goal g by returning a pair $(\alpha, Goal)$, where α is a ground action, and $Goal$ a conjunctive goal. If $Goal$ is empty, that means that α is an action that will make g true in the current situation. Otherwise, the planner subgoals to $Goal$:
3. **if** $Goal$ is empty **then** push α to Γ **else** FIND-PLAN($Goal$).
4. **until** g is true in the situation resulted from performing Γ in the initial situation.

Function SOLVE(g). This function determines how a simple goal g is to be achieved, either by an action α or by subgoalling to a conjunctive goal $Goal$. This is the main place where user can control the planner. If there are no user-provided control knowledge, then this function takes the following default steps:

1. **If** α is an action that is executable in the current situation and, once executed, will make g true, **then** return this α and let $Goal$ be empty. This is a backtracking point as there may be more than one action that can be immediately executed to make g true.
2. Let $\mathcal{A}$ be the set of ground actions that have g as one of their effects. Let $Goal$ be the conjunction of the simple goals that are part of the precondition of every action in $\mathcal{A}$.
3. **if** either $Goal$ is unachievable or any of conjunct of $Goal$ is already in the stack Π, **then** fail **else**
4. **if** $Goal$ is not yet true in the current situation **then** return with $Goal$ **else**
5. Find an action A that has g as one of its effects. In general, A may have some variables as only some of the parameters need to be instantiated to have g as one of its effects. This is a backtracking point as there may be more than one such actions.
6. Let $Goal$ be the conjunction of the ground members in A's precondition list.
7. **if** either $Goal$ is unachievable or any conjunct of $Goal$ is already in the stack Π, **then** fail **else**
8. **if** $Goal$ is not yet true in the current situation **then** return with $Goal$ **else**
9. Get a full instantiation of A, and let $Goal$ be the precondition list of this instantiation. This is a backtracking point as there may be more than one such instantiations.
10. **if** either $Goal$ is unachievable or any conjunct of $Goal$ is already in the stack Π, **then** fail **else**
11. **if** $Goal$ is not yet true in the current situation **then** return with $Goal$ **else**

Here the default strategy for checking whether a goal G is unachievable is very simple: if G has a subgoal that cannot be changed by any action in the domain, then for this subgoal to be achievable, it must be already true in the initial situation. Any more involved checking of unachievability must be provided by the user.

5 The blocks world

We now illustrate our planning algorithm with an example from the blocks world. In the version given at the AIPS-2000 competition, there are four actions, given below in STRIPS style:

```
pickup(x)
  PC: handempty, clear(x), ontable(x)
  D:  handempty, clear(x), ontable(x)
  A:  holding(x)
putdown(x)
  PC: holding(x)
  D:  holding(x)
  A:  handempty, clear(x), ontable(x)
stack(x,y)
  PC: holding(x), clear(y)
  D:  holding(x), clear(y)
  A:  on(x,y), clear(x), handempty
unstack(x,y)
  PC: handempty, on(x,y), clear(x)
  D:  handempty, on(x,y), clear(x)
  A:  holding(x), clear(y)
```

Our algorithm is sensitive to the order of actions. For now, we assume that the order is as above, which is the same order given at the AIPS-2000 competition. Now consider the following initial state description:

$$on(c,b), on(b,a), ontable(a), clear(c), handempty$$

and the goal $clear(a)$. Figure 1 shows the trace of our planning algorithm on this problem. Begin with the top goal $clear(a)$ in the $Goal$ column, it first pushes $clear(a)$ to the stack Π of simple goals at step 2 and proceeds to find a plan for this simple goal. Since there are no common preconditions among all the actions that can possibly make it true, it finds the first action that can make $clear(a)$ true, which is $putdown(a)$ in the $Action$ column, and makes the only (ground) precondition of this action, which is $holding(a)$, the new goal and records it in $Goal$ column. In step 4, $holding(a)$ is pushed to Π and becomes the current simple goal for the planner to achieve. In step 5, the planner finds out that among all actions that can possibly make $holding(a)$ true, $handempty, clear(a)$ are the common preconditions, so it makes their conjunction the new goal to achieve. But then it finds out that one of the conjuncts, $clear(a)$ is already in Π, so it fails the simple goal $holding(a)$ in step 6, and goes back to step 2 to find alternative actions for achieving $clear(a)$, which it finds to be $stack(a,x)$ for some x. But the ground precondition for this action is $holding(a)$ (as indicated in the $Goal$ column), which cannot possibly succeed as the previous steps shows, so it goes back to step 2 to find another alternative action for achieving $clear(a)$, and this time it finds $unstack(x,a)$ for some x in step 9. Since the common precondition of these actions is $handempty$ which is already true in the initial situation, so it begins to instantiate x according to some pre-set order on the blocks, which in our examples is $[a,b,c]$. Step 10 shows that $x=a$ will not work as one of the precondition of $unstack(a,a)$ would require $clear(a)$ which is already in Π. So it tries $x=b$ in step 12 and makes its preconditions, $handempty, on(b,a), clear(b)$, the new goal to achieve. Since the first two simple goals are already true in the current state, which is the initial state at this point, so it puts $clear(b)$ into Π. Now the planner tries to achieve $clear(b)$, and eventually returns the plan $[unstack(c,b), putdown(c), unstack(b,a)]$. Notice that in step 26, after $clear(b)$ is achieved with action $unstack(c,b)$, it remembers that this simple goal is part of the conjunctive goal in step 12, so it restores this conjunctive goal in the next step (step 27) as the current conjunctive goal to achieve and finds out that after this action, $handempty$ is no longer true, so it pushes it to Π in step 28. Step 29 returns $putdown(c)$ as the action for achieving this simple goal as it is the first action that is executable and will make it true once executed.

6 Post-Processing

While the planner returns a reasonable plan in the above example, this may not be the case in general. For instance,

	Actions	Π (Stack of simple goals)	Goal	Γ (Plans)
1.			clear(a)	
2.		clear(a)		
3.	putdown(a)	clear(a)	holding(a)	
4.		holding(a), clear(a)		
5.	pickup(a) or unstack(a,x)	holding(a), clear(a)	handempty, clear(a)	
6.		fail, go back to 2.		
7.	stack(a,x)	clear(a)	holding(a)	
8.		fail (like 3), go back to step 2.		
9.	unstack(x,a)	clear(a)	handempty	
10.	unstack(a,a)	clear(a)	handempty, on(a,a), clear(a)	
11.		fail, go back to step 9		
12.	unstack(b,a)	clear(a)	handempty, on(b,a), clear(b)	
13.		clear(b), clear(a)		
14.	putdown(b)	clear(b), clear(a)	holding(b)	
15.		holding(b), clear(b), clear(a)		
16.	pickup(b) or unstack(b,x)	holding(b), clear(b), clear(a)	handempty,clear(b)	
17.		fail, go back to 13.		
18.	stack(b,x)	clear(b), clear(a)	holding(b)	
19.		fail (like 16), go back to 13		
20.	unstack(x, b)	clear(b), clear(a)	handempty	
21.	unstack(a, b)	clear(b), clear(a)	handempty, on(a,b), clear(a)	
22.		fail, go back to 20		
23.	unstack(b, b)	clear(b), clear(a)	handempty, on(b,b), clear(b)	
24.		fail, go back to 20		
25.	unstack(c, b)	clear(b), clear(a)	handempty, on(c,b), clear(c)	
26.		clear(a)		unstack(c,b)
27.		clear(a)	handempty, on(b,a), clear(b) (from 12)	unstack(c,b)
28.		handempty, clear(a)		unstack(c,b)
29.		handempty, clear(a)		unstack(c,b), putdown(c)
30.		clear(a)	handempty, on(b,a), clear(b) (from 12)	unstack(c,b), putdown(c)
31.		clear(a)		unstack(c,b), putdown(c), unstack(b,a)

Figure 1: Execution trace for the goal clear(a).

given the same initial situation description, and the goal $on(a, b), on(b, c)$, it will return the following plan:

$$[unstack(c, b), putdown(c), unstack(b, a), putdown(b), pickup(a), stack(a, b), unstack(a, b), putdown(a), pickup(b), stack(b, c), pickup(a), stack(a, b)]$$

which is obviously not a very good one. For the AIPS-2000 competition, we implemented the following strategy for post-processing:

> Remove all immediate cycles: If an action is followed immediately by its complement, then remove both of the actions.

Here an action B is a complement of action A if B always restores whatever changes made by A. For instance, in the blocks world, $pickup(x)$ and $putdown(x)$ are complementary to each other, so are $stack(x, y)$ and $unstack(x, y)$. It is clear that this strategy is always correct and will return a plan that is at least as good as the original one. As it turned out, this is a very effective strategy for the blocks world. For instance, after post-processing, the above plan becomes:

$$[unstack(c, b), putdown(c), unstack(b, a), stack(b, c), pickup(a), stack(a, b)]$$

Table 1 shows the performance of our planner on some typical blocks world problems at the competition. Among 12 contenders in the fully automatic track, R performed the best in the blocks world, both in terms of the number of problems it solved and the length of the plans it returned. In fact, it is the only one that were able to solve all the problems within the time allowed.

However, effective as it is in the blocks world, this strategy is not very effective in the logistics domain. For instance, given a problem of moving, say two packages from a city to

Problem	Time	Plan length
20-0	9.26	72
25-0	27.46	90
30-0	67.41	104
35-0	146.63	128
40-0	286.96	146
45-0	523.09	174

Table 1: The blocks world. The first number in Problem column is the number of blocks in the problem; times are CPU seconds.

another city, our planner would generate a plan that moves a package from the starting city to the target city and then goes back to the original city to move the other package. There is little this strategy can do here as there are no immediate cycles. For this and other problems where there are many simple goals that do not interact with each other, we have found that the strategy of trying to perform an action as early as possible is very effective in maximizing potential concurrent actions. More precisely:

> Given an initial situation S, a goal G, and a plan $\alpha = [a_1, ..., a_n]$ for G in S, find a largest j, and a smallest i such that $j > i$, $\alpha' = [a_1, ..., a_{i-1}, a_j, a_i, ..., a_{j-1}, a_{j+1}, ..., a_n]$ is also a plan for G in S, and that $\{a_i, ..., a_j\}$ is not a legal concurrent action executable in the state resulted from performing $[a_1, ..., a_{i-1}]$ in S. Then replace α by α'.

Here a set of actions is a legal concurrent action if their effects do not interact, i.e. the union of their delete lists and that of their add lists do not intersect, and a legal concurrent action is executable in a state if every one of them in executable. Checking that $\{a_i, ..., a_j\}$ is not a legal concurrent action is necessary, because if, say $\{a_1, a_2\}$ is a legal concurrent action that can be executed in S, then they can be executed in any order, so there is no point for trying to move a_2 ahead of a_1.

Notice that replacing α by α' won't make the plan shorter right away. But the hope is that α' may have some immediate cycles that can be eliminated now. So the new strategy really is to combine the two operations, removing immediate cycles and promoting an action to an earlier time, do them one after the other until they produce no changes.

For instance, consider the following plan for moving packages $p1$ and $p2$ from city $c1$ to $c2$ using actions $m(x, y)$ - move the truck from x to y, $l(p)$ - load package p into the truck, and $u(p)$ - unload package p from the truck, assuming that the truck is initially in city $c1$:

$$[l(p1), m(c1, c2), u(p1), m(c2, c1), l(p2), m(c1, c2), u(p2)].$$

Clearly, there is no immediate cycles here. But we can move $l(p2)$ to the beginning of the plan:

$$[l(p2), l(p1), m(c1, c2), u(p1), m(c2, c1), m(c1, c2), u(p2)]$$

and this creates an immediate cycle. Removing it yields the following optimal plan:

$$[l(p2), l(p1), m(c1, c2), u(p1), u(p2)].$$

Notice here that although $l(p1)$ can also be the first action, we do not move it because $\{l(p2), l(p1)\}$ is a legal concurrent action that is executable in the initial state. Similarly, there is no point moving $u(p2)$ before $u(p1)$.

In general, post-processing is a difficult issue. Lin [1998] discusses some of the general strategies one can use to improve the quality of a plan. Ambite and Knoblock [1997] propose a formalism for expressing domain-specific post-processing strategies. While most of the current planners attempt to generate high quality plans in one shot, it is worthwhile to experiment with an alternative of trying to generate a plan very quickly without worry about the quality of the plans so generated, and then do some post-processing later. In a sense, this is what our planner tries to do.

7 Controlling the planner

The performance of this planner depends on the following factors:

- For each fluent F, the order of the actions that can make F true. For instance, in the blocks world, when attempting the simple goal $clear(x)$, a lot of backtracking can be avoided if we order $unstack(y, x)$ before other actions.
- The ordering of simple goals in a conjunctive goal, in particular, the ordering of preconditions of actions.

While it is easy to provide some mechanisms for the user to control these orderings explicitly, we do not do so for our planner because we believe this is not the level where the user wants to get her hands on, except that on top level goals, the user may have some knowledge about in what order they should be achieved, and in this case it can be done by a simple pre-processing step as illustrated below for the blocks world. In general, though, the burden should be on the planner to come up with a good ordering by analyzing the domain at hand. This is an area that we are working on for the next version of the planner.

The main mechanism that we provide the user with to control the planner with domain specific information is through a SOLVE-A-GOAL predicate that will override the default strategy used by the planner for trying to achieve a simple goal. Given a simple goal g, a situation s, SOLVE-A-GOAL$(g, Action, Goal, s)$ means that to achieve g in s, either do $Action$ or (when $Action = nil$) try to achieve $Goal$ first. For instance, for the blocks world, we can write in Prolog:

- To clear a block, clear the one that is on it:

```
solve-a-goal(clear(X), nil,[clear(Y)],s) :-
    holds([on(Y,X)],s),
    not holds([clear(Y)],s).
```

- To make the hand empty, drop the one that the robot is holding:

```
solve-a-goal(handempty, putdown(X), [],s) :-
    holds([holding(X)],s).
```

- To hold a block, clear it first (if it is already clear, there must be an action that is executable and can make it true, so it would not come to this step):

Problem	Time	Plan length
80-0	1.52	296
100-0	2.49	368
200-0	9.73	732
300-0	22.71	1158
500-0	65.35	1962

Table 2: The blocks world with control information: see Table 1 for legends.

Problem	Time	Plan length
logistics-16-0	44.94	175
logistics-19-0	77.82	180
logistics-22-0	192.89	219
logistics-25-0	376.47	350
logistics-28-0	719.33	424

Table 3: Logistics domain: fully automatic track.

solve-a-goal(holding(X), nil, [clear(X)],s).

These rules, together with a pre-processing step that would sort the top level goal according to the order:

$$[ontable(x_n), on(x_{n-1}, x_n), ..., on(x_2, x_1)],$$

are the control strategies used by our planner at the AIPS-2000 competition. Table 2 shows the performance of these control rules on some typical problems from the competition. One can also write similar rules to control the planner in other domains such as the logistics domain.

8 Concluding remarks

While STRIPS laid the foundation for much of the later work in AI planning, there had not been much follow-up work on the particular planning algorithm used in STRIPS, except for some of variants described in Nilsson [1980]. Compared to partial-order planners (see, e.g. [Sacerdoti, 1975; McAllester and Rosenblitt, 1991; Penberthy and Weld, 1992]), STRIPS and its variants are arguably simpler and easier to understand, thus more amenable to be controlled by the user with domain-specific information which is indispensable for large problems.

We have mentioned that our planner R performed very well in the blocks world IN the AIPS-2000 planning competition. It also performed well in the logistics domain, although not as well as some of the other contenders. Table 3 shows the performance of R on some of the logistics domain problems given in the competition, and Table 4 shows the performance of R on the same problems with some domain-specific control rules. The domain that our planner did the worst is the freecell domain, a popular poker game that comes with MS Windows.

An obvious question regarding this planner, and perhaps any other planners, is what its range of applicability is, i.e., in what domains it will be effective, and what domains it will not. While this is a theoretically challenging problem, it may not matter much in practice as simply running it on some of the problems in the domain will tell. A better question is whether there is another planner that will complements this one, i.e. it works well in domains that this one does not.

Problem	Time	Plan length
logistics-16-0	2.21	129
logistics-19-0	3.08	138
logistics-22-0	4.27	159
logistics-25-0	9.22	214
logistics-28-0	12.04	244

Table 4: Logistics domain: with hand-tailored control information.

There are some possible directions one can take to try to improve on the planner. As we mentioned, the performance of this planner depends crucially on the order of actions that one chooses to achieve a simple goal and on the order of simple goals in a conjunctive goal. We are currently working on strategies that will automatically discover some good action and goal orderings based on a systematic search in domains with small size.

References

[Ambite and Knoblock, 1997] J. Ambite and C. Knoblock. Planning by rewriting: efficiently generating high-quality plans. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI–97), AAAI Press, Menlo Park, CA.*, pages 706–713, 1997.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to theorem proving in problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Lin and Reiter, 1997] F. Lin and R. Reiter. How to progress a database. *Artificial Intelligence*, (92)1-2:131–167, 1997.

[Lin, 1998] F. Lin. On measuring plan quality. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 224–233, 1998.

[McAllester and Rosenblitt, 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI–91)*, pages 634–639, 1991.

[Nilsson, 1980] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA., 1980.

[Nilsson, 1998] N. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, Inc., San Francisco, CA., 1998.

[Penberthy and Weld, 1992] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, 1992.

[Sacerdoti, 1975] E. Sacerdoti. The nonlinear nature of plans. In *Proceedings of IJCAI-75*, pages 206–214, 1975.