Fast Planning Through Planning Graph Analysis*

Avrim L. Blum
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891
avrim@cs.cmu.edu

Merrick L. Furst
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891
mxf@cs.cmu.edu

(Final version in *Artificial Intelligence*, 90:281–300, 1997)

Abstract

We introduce a new approach to planning in STRIPS-like domains based on constructing and analyzing a compact structure we call a Planning Graph. We describe a new planner, Graphplan, that uses this paradigm. Graphplan always returns a shortest-possible partial-order plan, or states that no valid plan exists.

We provide empirical evidence in favor of this approach, showing that Graphplan outperforms the total-order planner, Prodigy, and the partial-order planner, UCPOP, on a variety of interesting natural and artificial planning problems. We also give empirical evidence that the plans produced by Graphplan are quite sensible. Since searches made by this approach are fundamentally different from the searches of other common planning methods, they provide a new perspective on the planning problem.

Keywords: General Purpose Planning, STRIPS Planning, Graph Algorithms, Planning Graph Analysis.

1 Introduction

In this paper we introduce a new planner, Graphplan, which plans in STRIPS-like domains. The algorithm is based on a paradigm we call Planning Graph Analysis. In this approach, rather than immediately embarking upon a search as in standard planning methods, the algorithm instead begins by explicitly constructing a compact structure we call a *Planning Graph*. A Planning Graph encodes the planning problem in such a way that many useful constraints inherent in the problem become explicitly available to reduce the amount of search needed. Furthermore, Planning Graphs can be constructed quickly: they have polynomial size and can be built in polynomial time. It is worth pointing out that a Planning

*This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The first author is also supported in part by NSF National Young Investigator grant CCR-9357793 and a Sloan Foundation Research Fellowship. The second author is supported in part by NSF grant CCR-9119319. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

Graph is *not* the state-space graph, which of course could be huge. In fact, unlike the state-space graph in which a plan is a *path* through the graph, in a Planning Graph a plan is essentially a *flow* in the network flow sense. Planning Graphs are closer in spirit to the Problem Space Graphs (PSGs) of Etzioni [1990], though unlike PSGs, Planning Graphs are based not only on domain information, but also the goals and initial conditions of a problem and an explicit notion of time.

Planning Graphs offer a means of organizing and maintaining search information that is reminiscent of the efficient solutions to Dynamic Programming problems. Planning Graph Analysis appears to have significant practical value in solving planning problems even though the inherent complexity of STRIPS planning, which is at least PSPACE-hard (e.g., see Bylander [1994]), is much greater than the complexity of standard Dynamic Programming problems. We provide empirical evidence on a variety of "natural" and artificial domains showing that Planning Graph Analysis is able to provide a quite substantial improvement in running time.

The Graphplan planner uses the Planning Graph that it creates to guide its search for a plan. The search that it performs combines aspects of both total-order and partial-order planners. Like traditional total-order planners, Graphplan makes strong commitments in its search. When it considers an action, it considers it at a specific point in time: for instance, it might consider placing the action `'move Rocket1 from London to Paris'` in a plan at exactly time-step 2. On the other hand, like partial-order planners [Chapman, 1987][McAllester and Rosenblitt, 1991][Barrett and Weld, 1994][Weld, 1994], Graphplan generates partially ordered plans. For instance, in Veloso's rocket problem (Figure 1), the plan that Graphplan finds is of the form: "In time-step 1, appropriately load all the objects into the rockets, in time-step 2 move the rockets, and in time-step 3, unload the rockets." The semantics of such a plan is that the actions in a given time step may be performed in any desired order. Conceptually this is a kind of "parallel" plan [Knoblock, 1994], since one could imagine executing the actions in three time steps if one had as many workers as needed to load and unload and fly the rockets.

One valuable feature of our algorithm is that it guarantees it will find the *shortest* plan among those in which independent actions may take place at the same time. Empirically and subjectively these sorts of plans seem particularly sensible. For example, in Stuart Russell's "flat-tire world" (the goal is to fix a flat tire and then return all the tools back to where they came from; see the UCPOP domains list), the plan produced by Graphplan opens the boot (trunk) in step 1, fetches all the tools and the spare tire in step 2, inflates the spare and loosens the nuts in step 3, and so forth until it finally closes the boot in step 12. (See Figure 4.) Another significant feature of our algorithm is that it is not particularly sensitive to the order of the goals in a planning task, unlike traditional approaches. More discussion of this issue is given in Section 3.2. In Section 4 of this paper we present empirical results that demonstrate the effectiveness of Graphplan on a variety of interesting "natural" and artificial domains.

An extended abstract of this work appears in [Blum and Furst, 1995].

1.1 Definitions and Notation

Planning Graph Analysis applies to STRIPS-like planning domains [Fikes and Nilsson, 1971]. In these domains, operators have preconditions, add-effects, and delete-effects, all of

The rocket domain (introduced by Veloso [1989]) has three operators: Load, Unload, and Move. A piece of cargo can be loaded into a rocket if the rocket and cargo are in the same location. A rocket may move if it has fuel, but performing the move operation uses up the fuel. In UCPOP format, the operators are:

```
(define (operator move)
    :parameters ((rocket ?r) (place ?from) (place ?to))
    :precondition (:and (:neq ?from ?to) (at ?r ?from) (has-fuel ?r))
    :effect (:and (at ?r ?to) (:not (at ?r ?from)) (:not (has-fuel ?r))))

(define (operator unload)
    :parameters ((rocket ?r) (place ?p) (cargo ?c))
    :precondition (:and (at ?r ?p) (in ?c ?r))
    :effect (:and (:not (in ?c ?r)) (at ?c ?p)))

(define (operator load)
    :parameters ((rocket ?r) (place ?p) (cargo ?c))
    :precondition (:and (at ?r ?p) (at ?c ?p))
    :effect (:and (:not (at ?c ?p)) (in ?c ?r)))
```

A typical problem might have one or more rockets and some cargo in a start location with a goal of moving the cargo to some number of destinations.

Figure 1: *A Simple Rocket Domain.*

which are conjuncts of propositions, and have parameters that can be instantiated to objects in the world. Operators do not create or destroy objects and time may be represented discretely. An example is given in Figure 1.

Specifically, by a *planning problem*, we mean:

- A STRIPS-like domain (a set of operators),
- A set of objects,
- A set of propositions (literals) called the Initial Conditions,
- A set of Problem Goals which are propositions that are required to be true at the end of a plan.

By an *action*, we mean a fully-instantiated operator. For instance, the operator `put ?x into ?y` may instantiate to the specific action `put Object1 into Container2`. An action taken at time t adds to the world all the propositions which are among its Add-Effects and deletes all the propositions which are among its Delete-Effects. It will be convenient to think of "doing nothing" to a proposition in a time step as a special kind of action we call a *no-op* or *frame* action.

2 Valid Plans and Planning Graphs

We now define what we mean when we say a set of actions forms a valid plan. In our framework, a *valid plan* for a planning problem consists of a set of actions and specified times in which each is to be carried out. There will be actions at time 1, actions at time 2, and so forth. Several actions may be specified to occur at the same time step so long as they do not interfere with each other. Specifically, we say that two actions *interfere* if one deletes a precondition or an add-effect of the other.[1] In a linear plan these *independent* parallel actions could be arranged in any order with exactly the same outcome. A valid plan may perform an action at time 1 if its preconditions are all in the Initial Conditions. A valid plan may perform an action at time $t > 1$ if the plan makes all its preconditions true at time t. Because we have no-op actions that carry truth forward in time, we may define a proposition to be true at time $t > 1$ if and only if it is an Add-Effect of some action taken at time $t - 1$. Finally, a valid plan must make all the Problem Goals true at the final time step.

2.1 Planning Graphs

A Planning Graph is similar to a valid plan, but without the requirement that the actions at a given time step not interfere. It is, in essence, a type of constraint graph that encodes the planning problem.

More precisely, a Planning Graph is a directed, leveled graph[2] with two kinds of nodes and three kinds of edges. The levels alternate between *proposition levels* containing *proposition nodes* (each labeled with some proposition) and *action levels* containing *action nodes* (each labeled with some action). The first level of a Planning Graph is a proposition level and consists of one node for each proposition in the Initial Conditions. The levels in a Planning Graph, from earliest to latest are: propositions true at time 1, possible actions at time 1, propositions possibly true at time 2, possible actions at time 2, propositions possibly true at time 3, and so forth.

Edges in a Planning Graph explicitly represent relations between actions and propositions. The action nodes in action-level i are connected by "precondition-edges" to their preconditions in proposition level i, by "add-edges" to their Add-Effects in proposition-level $i+1$, and by "delete-edges" to their Delete-Effects in proposition-level $i+1$.[3]

The conditions imposed on a Planning Graph are much weaker than those imposed on valid plans. Actions may exist at action-level i if all their preconditions exist at proposition-level i but there is no requirement of "independence." In particular, action-level i may

[1] Knoblock [1994] describes an interesting less restrictive notion in which several actions may occur at the same time even if one deletes an add-effect of another, so long as those add-effects are not important for reaching the goals.

[2] A graph is called *leveled* if its nodes can be partitioned into disjoint sets L_1, L_2, ..., L_n such that the edges only connect nodes in adjacent levels.

[3] A length-two path from an action a at one level, through a proposition Q at the next level, to an action b at the following level, is similar to a causal link $a \xrightarrow{Q} b$ in a partial-order planner.

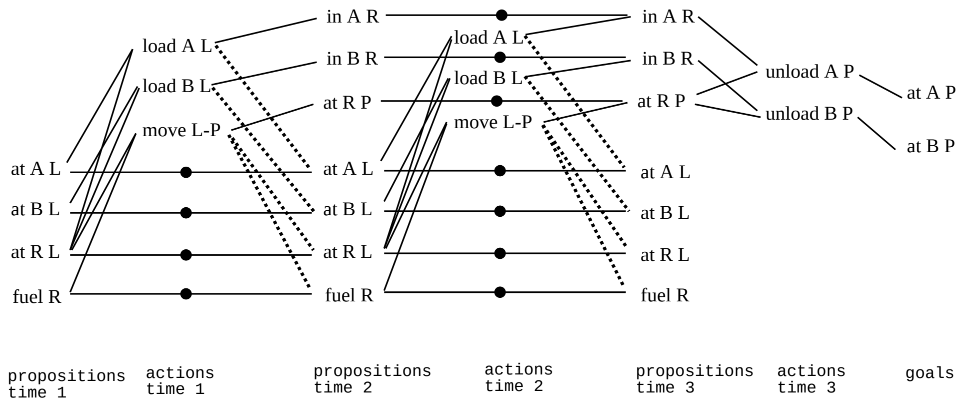

Figure 2: *A planning graph for the rocket problem with one rocket R, two pieces of cargo A and B, a start location L and one destination P. For simplicity, the "rocket" parameter has been removed from the actions' names. Delete edges are represented by dashed lines and no-ops are represented by dots. In the planning graph created by* Graphplan *for this problem, there would be more action nodes in the second and third action levels.*

legally contain *all* the possible actions whose preconditions all exist in proposition-level i. A proposition may exist at proposition-level $i+1$ if it is an Add-Effect of some action in action-level i (even if it is also a Delete-Effect of some other action in action-level i). Because we allow "no-op actions," every proposition that appears in proposition-level i may also appear in proposition-level $i+1$. An example of a Planning Graph is given in Figure 2.

Since the requirements on Planning Graphs are so weak, it is easy to create them. In Section 3.1 we describe how Graphplan constructs Planning Graphs from domains and problems. In particular, any Planning Graph with t action-levels that Graphplan creates will have the following property:

> If a valid plan exists using t or fewer time steps, then that plan exists as a subgraph of the Planning Graph.

It is worth noting here that Planning Graphs are not overly large. See Theorem 1.

2.2 Exclusion Relations Among Planning Graph Nodes

An integral part of Planning-Graph Analysis is noticing and propagating certain *mutual exclusion* relations among nodes. Two actions at a given action level in a Planning Graph are *mutually exclusive* if no valid plan could possibly contain both. Similarly, two propositions at a given proposition level are mutually exclusive if no valid plan could possibly make both true. Identifying mutual exclusion relationships can be of enormous help in reducing the search for a subgraph of a Planning Graph that might correspond to a valid plan.

Graphplan notices and records mutual exclusion relationships by propagating them through the Planning Graph using a few simple rules. These rules do not guarantee to find *all* mutual exclusion relationships, but usually find a large number of them.[4] Specifically, there

[4] In fact, determining *all* mutual exclusion relationships can be as hard as finding a legal plan. For

are two ways in which actions a and b at a given action-level are marked by Graphplan to be exclusive of each other:

[Interference] If either of the actions deletes a precondition or Add-Effect of the other. (This is just the standard notion of "non independence" and depends only on the operator definitions.)

[Competing Needs] If there is a precondition of action a and a precondition of action b that are marked as mutually exclusive of each other in the previous proposition level.

Two propositions p and q in a proposition-level are marked as exclusive if all ways of creating proposition p are exclusive of all ways of creating proposition q. Specifically, they are marked as exclusive if each action a having an add-edge to proposition p is marked as exclusive of each action b having an add-edge to proposition q.

For instance, in the rocket domain with `'Rocket1 at London'` and `'has-fuel Rocket1'` in the Initial Conditions, the actions `'move Rocket1 from London to Paris'` and `'load Alex into Rocket1 in London'` at time 1 are exclusive because the first deletes the proposition `'Rocket1 at London'` which is a precondition of the second. The proposition `'Rocket1 at London'` and the proposition `'Rocket1 at Paris'` are exclusive at time 2 because all ways of generating the first (there is only one: a no-op) are exclusive of all ways of generating the second (there is only one: by moving). The actions `'load Alex into Rocket1 in London'` and `'load Jason into Rocket1 in Paris'` (assuming we defined the initial conditions to have Jason in Paris) at time 2 are exclusive because they have competing needs, namely the propositions `'Rocket1 at London'` and `'Rocket1 at Paris'`.

A pair of propositions may be exclusive of each other at every level in a planning graph or they may start out being exclusive of each other in early levels and then become non-exclusive at later levels. For instance, if we begin with Alex and Rocket1 at London (and they are nowhere else at time 1), then `'Alex in Rocket1'` and `'Rocket1 at Paris'` are exclusive at time 2, but not at time 3.

2.2.1 The power of exclusion relations

Note that the Competing Needs notion and the exclusivity between propositions are not just logical properties of the operators. Rather, they depend on the interplay between operators and the Initial Conditions.

Consider, for instance, a domain such as the Rocket domain having a move operator. The useful notion that an item cannot be in two places at the same time is not just a function of the operators; if the initial conditions specified that the item started out in two different places, then it *could* continue to be in two places at once. Instead this notion depends both on the definition of `'move'` and the fact that the item starts out in only one place. The mutual exclusion rules provide a mechanism for propagating this notion through the graph. The reason is that if at time $t-1$ you can be in only one place, then any two move actions you might perform at time $t-1$ will be exclusive (any two moves from *different* starting locations are exclusive by Competing Needs and two moves from the *same* starting

instance, consider creating two new artificial goals g_1 and g_2 such that satisfying g_1 requires satisfying half of the original goals and satisfying g_2 requires satisfying the other half. Then, determining whether g_1 and g_2 are mutually exclusive is equivalent to solving the planning problem.

location are exclusive since they delete each others' preconditions) and therefore you can be in only one place at time t. Propagating these constraints allows the system to use this important fact in planning.

More generally, in many different domains, exclusion relations seem to propagate a variety of intuitively useful facts about the problem throughout the graph.

3 Description of the algorithm

The high-level description of our basic algorithm is the following. Starting with a Planning Graph that only has a single proposition level containing the Initial Conditions, Graphplan runs in stages. In stage i Graphplan takes the Planning Graph from stage $i-1$, extends it one time step (the next action level and the following proposition level), and then searches the extended Planning Graph for a valid plan of length i. Graphplan's search either finds a valid plan (in which case it halts) or else determines that the goals are not all achievable by time i (in which case it goes on to the next stage). Thus, in each iteration through this Extend/Search loop, the algorithm either discovers a plan or else proves that no plan having that many time steps or fewer is possible.

Graphplan's algorithm is sound and complete: any plan the algorithm finds is a legal plan, and if there exists a legal plan then Graphplan will find one. In Section 5 we describe how this algorithm may be augmented so that if the Problem Goals are not satisfiable by *any* valid plan, then the planner is guaranteed to halt with failure in finite time. This termination guarantee is one that is not provided by most partial-order planners.

3.1 Extending Planning Graphs

All the initial conditions are placed in the first proposition level of the graph. To create a generic action level, we do the following. For each operator and each way of instantiating preconditions of that operator to propositions in the previous level, insert an action node *if no two of its preconditions are labeled as mutually exclusive.*[5] Also insert all the no-op actions and insert the precondition edges. Then check the action nodes for exclusivity as described in Section 2.2 above and create an "actions-that-I-am-exclusive-of" list for each action.

To create a generic proposition level, simply look at all the Add-Effects of the actions in the previous level (including no-ops) and place them in the next level as propositions, connecting them via the appropriate add and delete-edges. Mark two propositions as exclusive if all ways of generating the first are exclusive of all ways of generating the second.

As we demonstrate in the following theorem, the time taken by our algorithm to create this graph structure is polynomial in the length of the problem's description and the number of time steps.

Theorem 1 *Consider a planning problem with n objects, p propositions in the Initial Conditions, and m STRIPS operators each having a constant number of formal parameters. Let*

[5] Checking for exclusions keeps Graphplan, for instance, from inserting the action `'unload Alex from Rocket1 in Paris'` in time 2 of the rocket-domain graph when the initial conditions specify that both Alex and the rocket begin in London.

ℓ be the length of the longest add-list of any of the operators. Then, the size of a t-level planning graph created by Graphplan*, and the time needed to create the graph, are polynomial n, m, p, ℓ, and t.*

Proof. Let k be the largest number of formal parameters in any operator. Since operators cannot create new objects, the number of different propositions that can be created by instantiating an operator is $O(\ell n^k)$. So, the maximum number of nodes in any proposition-level of the planning graph is $O(p + m\ell n^k)$. Since any operator can be instantiated in at most $O(n^k)$ distinct ways, the maximum number of nodes in any action-level of the planning graph is $O(mn^k)$. Thus the total size of the planning graph is polynomial in n, m, p, ℓ, and t, since k is constant.

The time needed to create a new action and proposition level of the graph can be broken down into (A) the time to instantiate the operators in all possible ways to preconditions in the previous proposition-level, (B) the time to determine mutual exclusion relations between actions, and (C) the time to determine the mutual exclusion relations in the next level of propositions. It is clear that this time is polynomial in the number of nodes in the current level of the graph. ■

Empirically, the part of graph creation that takes the most time is determining exclusion relations. However, empirically, graph creation only takes up a significant portion of Graphplan's running time in the simpler problems, where the total running time is not very large anyway.

An obvious improvement to the basic algorithm described above (which is implemented in Graphplan) is to avoid searching until a proposition-level has been created in which all the Problem Goals appear and no pair of Problem Goals has been determined to be mutually exclusive.

3.2 Searching for a plan

Given a Planning Graph, Graphplan searches for a valid plan using a backward-chaining strategy. Unlike most other planners, however, it uses a level-by-level approach, in order to best make use of the mutual exclusion constraints. In particular, given a set of goals at time t, it attempts to find a set of actions (no-ops included) at time $t-1$ having these goals as add effects. The preconditions to these actions form a set of subgoals at time $t-1$ having the property that if *these* goals can be achieved in $t-1$ steps, then the original goals can be achieved in t steps. If the goal set at time $t-1$ turns out not to be solvable, Graphplan tries to find a different set of actions, continuing until it either succeeds or has proven that the original set of goals is not solvable at time t.

In order to implement this strategy, Graphplan uses the following recursive search method. For each goal at time t in some arbitrary order, select some action at time $t-1$ achieving that goal that is not exclusive of any actions that have already been selected. Continue recursively with the next goal at time t. (Of course, if by good fortune a goal has already been achieved by some previously-selected action, we do not need to select a new action for it.) If our recursive call returns failure, then try a different action achieving our current goal, and so forth, returning failure once all such actions have been tried. Once finished with all the goals at time t, the preconditions to the selected actions make up the new goal set at time $t-1$. We call this a "goal-set creation step." Graphplan then continues this procedure at time step $t-1$.

A "forward-checking" improvement to this approach (which is implemented in Graphplan and helps modestly in our experiments) is that after each action is considered a check is made to make sure that no goal ahead in the list has been "cut-off." In other words, Graphplan checks to see if for some goal still ahead in the list, all the actions creating it are exclusive of actions we have currently selected. If there is some such goal, then Graphplan knows it needs to back up right away.

3.2.1 Memoization

One additional aspect of Graphplan's search is that when a set of (sub)goals at some time t is determined to be not solvable, then before popping back in the recursion it *memoizes* what it has learned, storing the goal set and the time t in a hash table. Similarly, when it creates a set of subgoals at some time t, before searching it first probes the hash table to see if the set has already been proved unsolvable. If so, it then backs up right away without searching further. This memoizing step, in addition to its use in speeding up search, is needed for our termination check described in Section 5.

3.2.2 An example

To make this more concrete, let us consider again the rocket problem in which the Initial Conditions have two fueled rockets and n pieces of cargo at some starting location S and the goal is to move some of the cargo to location X and some to location Y. For this problem, the graph will grow to contain three action levels. The planner will then select some goal, say `'A at X'`, and pick some action at time step 3 such as `'unload A from Rocket1 at X'` making it true. It then marks as not-doable all actions exclusive of this one, such as `'unload C from Rocket1 at Y'`, at time step 3. The planner then selects the next goal, say `'B at X'`. If it chooses to make this goal true by performing `'unload B from Rocket2 at X'` at time 3, then it will notice that a goal such as `'C at Y'` further down in its goal list has been completely cut off, because all ways of making it true are exclusive of the actions already committed to. Thus, Graphplan will instead select `'unload B from Rocket1 at X'`, and so on. Once the planner is done with all goals at this level, it then creates a new goal-set at the previous time step consisting of goals such as `'A in Rocket1'` and `'Rocket1 at X'` that were the preconditions of the actions selected.

3.2.3 The limited effect of goal orderings

The strategy of working on the subgoals in a somewhat breadth-first-like manner makes Graphplan fairly insensitive to goal-orderings. We now add one final feature to Graphplan's search strategy that will allow us to make this statement more precise. Let G be a goal set at some time t. We say that a non-exclusive set of actions A at time $t-1$ is a *minimal set of actions achieving* G if (1) every goal in G is an add-effect of some action in A, and (2) no action can be removed from A so that the add effects of the actions remaining still contain G. The modification to Graphplan's strategy is to only recurse on minimal action sets. If the set of actions A chosen by Graphplan to achieve some goal-set G is *not* minimal, we back up right away. (For instance, say our goals are g_1 and g_2; we pick some action achieving g_1 and then the action we choose to achieve g_2 happens to also achieve g_1 as well. This would not be minimal.) This modification allows us to make a clean statement about the goal-sets that Graphplan considers. Specifically, we can state the following theorem.

Theorem 2 *Let G be a goal set at some time t that is* not *solvable in t steps. Then, no matter what the ordering of the goals in G, the goal sets at time $t-1$ that* Graphplan *considers when attempting to achieve G are exactly the preconditions of all the minimal action sets at time $t-1$ achieving G. (If G* is *solvable in t steps, then* Graphplan *may halt before considering all those goal sets).*

Proof. We have forced Graphplan to consider *only* minimal action sets; we need to show that *every* such set is examined. Let A be some such set, and consider some arbitrary ordering of G. Let a_1 be some action in A achieving the first goal in G (and let's call that goal g_{a_1}). Let a_2 be the action in A achieving the first goal in G not already achieved by a_1 (and let's call that goal g_{a_2}). More generally, let a_i be the action in A achieving the first goal in G not achieved by any of $\{a_1, \ldots, a_{i-1}\}$, and we will call that goal g_{a_i}. Notice that all actions in A are given an index in this way because A is minimal. This ordering of the actions implies that at some point in the recursion, a_1 will be the action chosen by Graphplan to achieve goal g_{a_1}; given that that occurs, at some point a_2 will be the action chosen to achieve g_{a_2}, and so forth. Therefore, all actions in A are considered. ∎

We can now quantify the limited effect of goal ordering as follows. Suppose Graphplan is currently attempting to solve the Problem Goals at some time T and is unsuccessful. Then, the total number of goal-sets examined in the search is *completely independent of* the ordering of the goals. The effect of goal ordering is limited to (A) the amount of time it takes on average to examine a new goal set (perform a goal-set creation step), and (B) the amount of work performed in the final stage at which the Problem Goals *are* found to be solvable (since goal ordering may affect the order in which goal sets are examined). In addition to this theoretical statement, empirically, Graphplan's dependence on goal ordering seems to be quite small: significantly less than that of other planners such as Prodigy and UCPOP.

4 Experimental Results

4.1 Natural domains

We compared Graphplan with two popular planners, Prodigy and UCPOP, on several "natural" planning problems from the planning literature. We ran Prodigy with heuristics suggested in Stone et al. [1994] and by Carbonell [Carbonell, personal communication]. It is somewhat unfair to compare exact running times because the planners are written in different languages (Graphplan is written in C while the other planners are in compiled Lisp), though partly because of this we ran Prodigy and UCPOP on a faster machine with more memory: we ran graphplan on a DECstation 2100 and the other planners on a SPARC10. Nonetheless, we can gain useful information from the curvature of plots of problems size versus time, as well as by comparing other objective measures. In particular, in addition to running time, we also report for Graphplan the number of goal-set creation steps (the number of times it creates a goal set at time $t-1$ from a goal set at time t) and the total

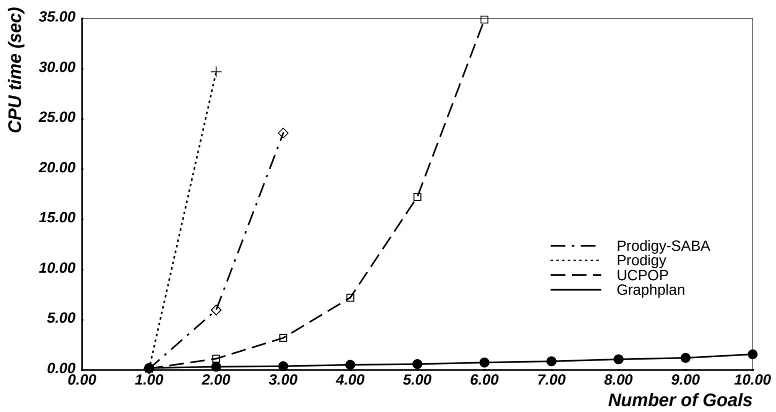

Figure 3: *2-Rockets problem*

number of times that it selects a non-noop action to try in its search. These are somewhat analogous to the backward-chaining steps taken by total-order planners.

4.1.1 Rocket

We ran the planners on the rocket domain described in Figure 1 with the following setup. The initial conditions have 3 locations (London, Paris, JFK), two rockets, and n items of cargo. All the objects (rockets and cargo) begin at London and the rockets have fuel. The goal is to get $\lceil n/2 \rceil$ of the objects to Paris and $\lfloor n/2 \rfloor$ of the objects to JFK. The goals are ordered alternating between destinations.

Results of the experiment are in Figure 3. Notice that Graphplan significantly outperforms the other two planners on this domain. Graphplan does well in this domain for two main reasons: (1) the Planning Graph only grows to 3 time steps, and (2) the mutual exclusion relations allow a small number of commitments (unloading something from Rocket1 in Paris and something else from Rocket2 in JFK) to completely force the remainder of the decisions. In particular, Graphplan performs only two goal-set creation steps regardless of the number of goals, and the number of non-noop actions tried is linear in the number of goals. The size of the graph created is also linear in the number of goals: there are 150 nodes total for the problem with two goals, and 37 additional nodes per goal from then on.

The running time of Graphplan is completely unaffected by goal ordering for this problem.

4.1.2 Flat Tire

A natural problem of a different sort is Stuart Russell's "fixing a flat tire" scenario (domain `init-flat-tire`, problem `fixit` in the UCPOP distribution). Unlike the rocket domain, a valid plan for solving this problem requires at least 12 time steps (and 19 actions). While for the rocket domain, Graphplan would do pretty well even without the mutual exclusion propagation, here the mutual exclusions are critical and ensure that not too many goal sets will be examined. Graphplan solves this problem in 1.1 to 1.3 seconds depending on the goal ordering. The number of goal-set creation steps ranges from a minimum of 105 to a maximum of 246, and the number of non-noop actions tried ranges from 170 to 350. The final graph created contains 786 nodes. Neither UCPOP nor Prodigy found a solution

Step 1:	open boot
Step 2:	fetch wrench boot fetch pump boot fetch jack boot fetch wheel2 boot
Step 3:	inflate wheel2 loosen nuts the-hub
Step 4:	put-away pump boot jack-up the-hub
Step 5:	undo nuts the-hub
Step 6:	remove-wheel wheel1 the-hub
Step 7:	put-on-wheel wheel2 the-hub put-away wheel1 boot
Step 8:	do-up nuts the-hub
Step 9:	jack-down the-hub
Step 10:	put-away jack boot tighten nuts the-hub
Step 11:	put-away wrench boot
Step 12:	close boot

Figure 4: Graphplan*'s plan for Russell's "Fixit" problem.*

within 10 minutes for this problem in the standard goal ordering, though it is possible to find goal orderings where they succeed much more quickly. Graphplan is not only fast on this domain, but also by producing the shortest partial-order plan, its plan is intuitively "sensible". Figure 4 shows the plan produced by Graphplan for this problem.

4.1.3 Monkey and Bananas

The UCPOP distribution provides three "Monkey and Bananas" problems (originally from Prodigy). Two have a solution and the third does not. Srinivasan and Howe [Srinivasan and Howe, 1995] show experimental results for a variety of partial-order planning heuristics on this domain. They report average running times (on a SPARC IPX, in Common Lisp) of about 90 seconds for most of the methods, though one took 2000 seconds and one took only 30 seconds on average per problem. They report an average number of plans examined in those planners for a task called "flaw selection" ranging from 5,558 to 105,518. Graphplan solves these problems much more quickly, taking 0.7 seconds on the first, 3.4 seconds on the second, and 2.8 seconds on the unsolvable one (these times are on a DECstation 2100). Graphplan attempts only 6 non-noop actions in solving the first problem, and 90 on the second. On the unsolvable problem, Graphplan extends its graph to 7 time steps, at which point it notices that the problem is unsolvable because the graph has "leveled off" and yet there still remain exclusive goals (see Section 5). Thus, on this problem Graphplan is able to report that the problem is unsolvable without actually performing any search.

On all three problems, most of the time spent is in graph creation. The graphs for the three problems contain 304, 824, and 700 nodes, respectively.

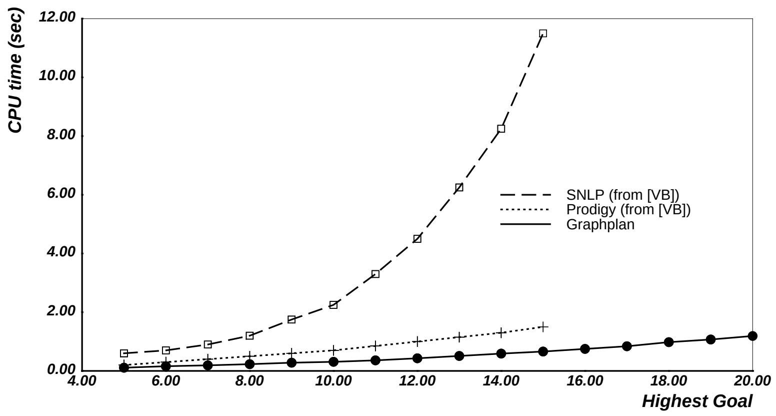

Figure 5: *Link-repeat domain from (Veloso & Blythe 1994)*

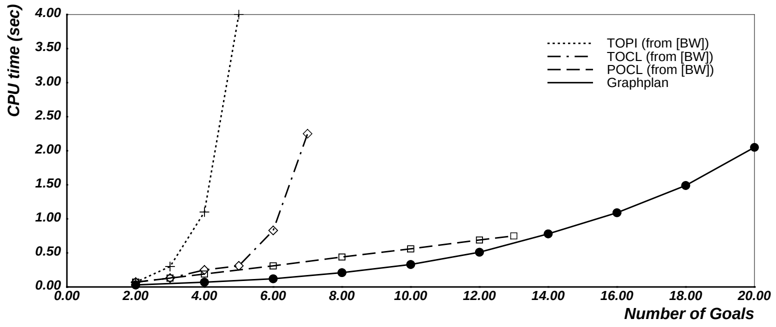

Figure 6: D^1S^1 *domain from (Barrett & Weld 1994)*

4.1.4 The Fridge Domain

The UCPOP distribution provides two "refrigerator fixing" domains. On the first one, Graphplan takes 4.0 seconds, performs 2 goal-set creation steps, and attempts 7 non-noop actions. On the second one Graphplan takes 11.3 seconds, performs 46 goal-set creation steps, and attempts 258 actions. On these two problems, the graphs created contain 287 and 686 nodes, respectively.

Srinivasan and Howe [1995] report times ranging from 30 to 300 seconds and average number of plans examined from about 9700 to 42000 for the different methods they consider.

4.2 Artificial domains

Barrett and Weld [1994] and Veloso and Blythe [1994] define a collection of artificial domains intended to distinguish the performance characteristics of various planners. On all of these, Graphplan is quite competitive with the best performance reported.

We present in Figures 5, 6, 7, and 8 performance data on four of the more interesting domains. All performance results in these figures for the other planners are taken from figures in their respective papers. (Note: in Figure 8, the TOCL and POCL curves effectively coincide.)

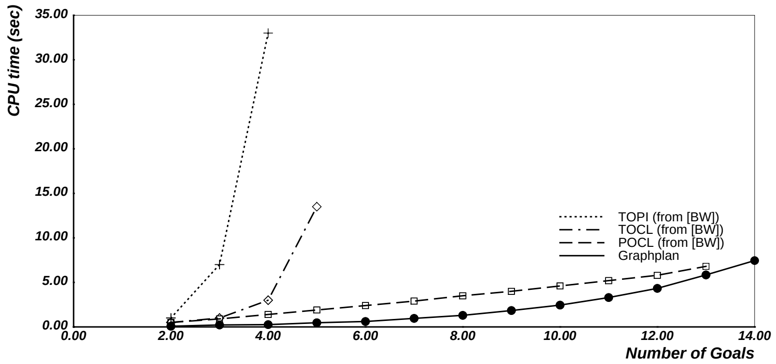

Figure 7: D^1S^2 *domain from (Barrett & Weld 1994)*

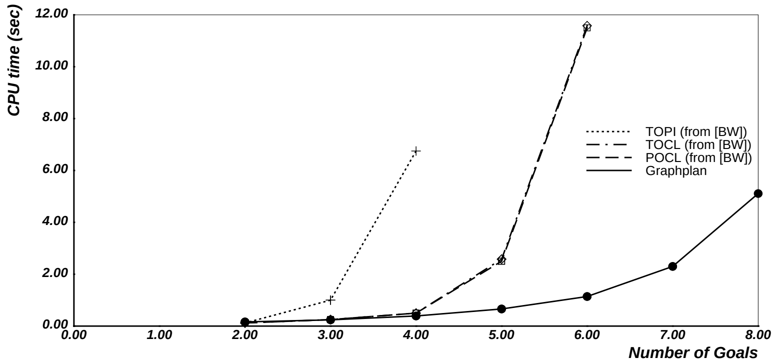

Figure 8: D^mS^{2*} *domain from (Barrett & Weld 1994)*

4.3 Discussion of Experimental Results

Four major factors seem to account for most of Graphplan's efficiency. They are, in order of empirically-derived importance:

Mutual Exclusion: In many of the examples, the pairwise mutual exclusions relations are able to represent most of the important constraints in the planning problem. (E.g., see the discussion in Section 2.2.1.) Propagating these constraints effectively prunes a large part of the search space.

Consideration of Parallel Plans: In some cases, such as the rocket problem, the valid parallel plans are relatively short compared with the length of the corresponding totally-ordered plans. In such cases neither the cost of Planning Graph construction, nor the cost of search is very large.

Memoizing: By fixing actions at specific points in time, Graphplan is able to record the goal sets that it proves to be unreachable in a certain number of time steps from the initial conditions.

Low-level costs: By constructing a Planning Graph in advance of search, Graphplan avoids the costs of performing instantiations during the searching phase.

Furthermore, it is worth noting that graph creation is quite fast (as well as being provably polynomial time) and only takes up a significant fraction of the total time on the simpler problems where the total running time is quite short in any case.

5 Terminating on Unsolvable Problems

To a first approximation, Graphplan conducts something like an iteratively-deepened search. In the i^{th} stage the algorithm sees if there is a valid parallel plan of length less than or equal to i. As described so far, if no valid plan exists there is nothing that prevents the algorithm from mindlessly running forever through an infinite number of stages.

We now describe a simple and efficient test that can be added after every unsuccessful stage so that if the problem has no solution then Graphplan will eventually halt and say "No Plan Exists."

5.1 Planning Graphs "Level Off"

Assume a problem has no valid plan. First observe that in the sequence of Planning Graphs created there will eventually be a proposition level P such that all future proposition levels are exactly the same as P, *i.e.*, they contain the same set of propositions and have the same exclusivity relations.

The reason for this is as follows. Because of the no-op actions, if a proposition appears in some proposition level then it also appears in all future proposition levels. Since only a finite set of propositions can be created by STRIPS-style operators (when applied to a finite set of initial conditions) there must be some proposition level Q such that all future levels have exactly the same set of propositions as Q. Also, again because of the no-op actions, if propositions p and q appear together in some level and are *not* marked as mutually exclusive, then they will not be marked as mutually exclusive in any future level. Thus there must be some proposition level P after Q such that all future proposition levels also have exactly the same set of mutual exclusion relations as P.

In fact, it is not hard to see that once two adjacent levels P_n, P_{n+1} are identical, then all future levels will be identical to P_n as well. At this point, we say the graph has *leveled off.*

5.2 A Quick and Easy Test

Let P_n be the first proposition level at which the graph has leveled off. If some Problem Goal does not appear in this level, or if two Problem Goals are marked as mutually exclusive in this level, then Graphplan can immediately say that no plan exists. Notice that in this case, Graphplan is able to halt without performing any search at all. This is what happened with the unsolvable "Monkey and Bananas" problem discussed in Section 4.1.3. However, in some cases it may be that no plan exists but this simple test does not detect it. A nice example of this is a blocks world with three blocks, in which the goals are for block A to be on top of block B, block B to be on top of block C, and block C to be on top of block A; any two of these goals are achievable but not all three simultaneously. So we need to do something slightly more sophisticated to guarantee termination in all cases.

5.3 A Test to Guarantee Termination

As mentioned earlier, Graphplan memoizes, or records, goal sets that it has considered at some level and determined to be unsolvable. Let S_i^t be the collection of all such sets stored for level i after an unsuccessful stage t. In other words, after an unsuccessful stage t, Graphplan has determined two things: (1) any plan of t or fewer steps must make one of the goal sets in S_i^t true at time i, and (2) none of the goal sets in S_i^t are achievable in i steps. The modification to Graphplan ensure termination is now just the following:

> If the graph has leveled off at some level n and a stage t has passed in which $|S_n^{t-1}| = |S_n^t|$, then output "No Plan Exists."

Theorem 3 Graphplan *outputs "No Plan Exists" if and only if the problem is unsolvable.*

Proof. The easy direction is that if the problem is unsolvable, then Graphplan will eventually say that no plan exists. The reason is just that the number of sets in S_n^t is never *smaller* than the number of sets in S_n^{t-1}, and there is a finite maximum (though exponential in the number of nodes at level n).

To see the other direction, suppose the graph has leveled off at some level n and Graphplan has completed an unsuccessful stage $t > n$. Notice that any plan to achieve some set in S_{n+1}^t must, one step earlier, achieve some set in S_n^t. This is because of the way Graphplan works: it determined each set in S_{n+1}^t was unsolvable by mapping it to sets at time step n and determining that they were unsolvable. Notice also that since the graph has leveled off, $S_{n+1}^t = S_n^{t-1}$. That is because the last $t-n$ levels of the graph are the same no matter how many additional levels the graph has.

Now suppose that after an unsuccessful stage t, $|S_n^{t-1}| = |S_n^t|$ (which implies that $S_n^{t-1} = S_n^t$). This means that $S_{n+1}^t = S_n^t$. Thus, in order to achieve any set in S_{n+1}^t one must previously have achieved some other set in S^t_{n+1}. Since none of the sets in S^t_{n+1} are contained in the initial conditions, the problem is unsolvable. ■

6 Additional Features

We have discussed so far the basic algorithm used by Graphplan. We now describe a few additional features that can be added in a natural way (and have been added as options in our implementation), and discuss their significance.

The first feature is a type of reasoning that is quite natural in our framework. The reasoning is that if the current goal set contains n goals such that no two of them can be made true at the same time by a non-noop action (and none of them are present in the Initial Conditions), then any plan will require at least n steps. For instance, one could use this reasoning in a path-finding domain to show that it must take at least n steps to visit n distinct places. Unfortunately, finding the largest such subset of any given goal set is equivalent to the maximum Clique problem (think of there being a "can't both be created now" edge between any two propositions that cannot both be made true in the same step). However, we can find a *maximal* such set using greedy methods.

This form of reasoning turns out to be very useful on traveling-salesman-like problems, where the goal is to visit all the nodes in a graph in as few steps as possible. On very dense graphs (such as the complete graph) for which the problem should be easy, Graphplan without this reasoning can be quite slow because the pairwise exclusion relations do not propagate well. For instance, on a complete graph, after two time steps any two goals of the form `'visited X'` will be non-exclusive. However, with this reasoning, Graphplan's performance is more respectable.

A second feature concerns graph creation. Although, as demonstrated in Theorem 1, the graph size is polynomial, it may be unnecessarily large if there are many irrelevant facts in the initial conditions. One way around this problem is to begin with a regression analysis going backward from the goals to determine if any initial conditions may be thrown out. For instance, if our rocket problem contains in the initial conditions a "junkyard" of rockets with no fuel, or some number of irrelevant observers, this method can identify them and set them aside. Of course, performing this regression analysis itself takes some amount of time.

One final feature (not currently in our implementation) that could be added easily is the ability to use the information learned on one planning problem for another problem on the same domain having the same Initial Conditions. Specifically, the same graph and the same memoized unsolvable goal sets could be re-used in this case.

7 Discussion and Future Work

We have described a novel planning algorithm, Graphplan. This algorithm uses ideas from standard total-order and partial-order planners, but differs most significantly by taking the position that representing the planning problem in a graph structure — a structure one can analyze, annotate, and play with — can significantly improve efficiency. Performance on the problems we have tried indicate that indeed this can provide a big savings.

We believe that even more significant gains will come from combining the approach of Graphplan with ideas, heuristics, and learning methods that have been developed in the planning literature. Specifically, directions we are currently considering include:

Learning: Learning techniques found to be useful for other planning methods (e.g., [Etzioni, 1990]) may work here as well. In addition, perhaps the new representation used here will suggest other learning approaches not considered previously.

Symmetry detection: Many of the times that planners behave poorly are times when symmetries exist in a problem that the planner does not utilize. Representing the planning problem as a graph may allow for new methods of detecting symmetries that could drastically reduce the search needed.

Two-way searches: Some problems are more easily solved in the forward direction than in the reverse. Prodigy, for instance, is able to create a plan in a forward direction even while it searches from the goals. We would like to incorporate some method for planning in a similar manner. This might involve memoizing solvable goal sets as well as unsolvable ones.

Other information to propagate: Graphplan propagates pairwise exclusion relations in order to speed up its search. There may be other sorts of information that could be propagated forward or backward through the graph that would be useful as well.

Using max-flow algorithms: The original motivation for our approach was that planning graphs, with slight modification, allow one to think of planning as a certain kind of maximum flow problem.[6] The view of planning as a flow problem requires additional constraints that make the problem NP-hard (in particular, a constraint that certain edges be either unused or else fully saturated, corresponding to the fact that an action may be performed or not performed, but cannot be "partially performed"). Nonetheless, perhaps algorithms for the max-flow problem — and there are many fast algorithms known [Cormen *et al.*, 1990, Goldberg and Tarjan, 1986] — might be useful for guiding the planning process. Empirically, we found that an approach based solely on max-flow algorithms did not perform as well as the method of backward-chaining with mutual exclusion relations described in this paper. A flow-based method, however, may allow one to naturally incorporate other aspects of a planning problem, such as having different costs associated with different actions, in a natural way. We are currently exploring whether flow algorithms can be *combined* with our current approach to improve performance.

7.1 Limitations and Open Problems

One main limitation of Graphplan is that it applies only to STRIPS-like domains. In particular, actions cannot create new objects and the effect of performing an action must be something that can be determined statically. There are many kinds of planning situations

[6] In this problem, one is given a graph containing source and sink nodes, and each edge is labeled with a capacity representing the maximum amount of fluid that may flow across that edge. In a legal flow, for every node except the source or sink, the flow in must equal the flow out. The goal is to flow as much fluid as possible from the source to the sink, without exceeding any of the capacities.

that violate these conditions. For instance, if one of the actions allows the planner to dig a hole of an arbitrary integral depth, then there are a potentially infinitely many objects that can be created. Or, suppose we have the action "paint everything in this room red." The effect of this action cannot be determined statically: the set of objects painted red depends on which happen to be in the room at the time. One open question is whether the Planning Graph Analysis paradigm can be extended to handle settings with these sorts of actions.

A second limitation is that roughly, in order to perform well Graphplan requires either that the pairwise mutual exclusion relations capture important constraints of the problem, or else that the ability to perform parallel actions significantly reduces the depth of the graph. Luckily, it appears that at least one of these tends to be true in many natural problems. Section 6 discussed one case (a simple TSP problem), however, in which neither of these occurs and Graphplan performs poorly without extra ad-hoc reasoning capabilities. Perhaps additional more powerful types of constraints can be added to Graphplan to overcome some of these difficult cases.

Finally, one last limitation worth mentioning is that by guaranteeing to find the *shortest* possible plan, Graphplan can make problems more difficult for itself. For instance, when people solve the 16-puzzle, they usually do so using a methodical approach that is easy to perform, but does not guarantee the solution with the fewest moves. If one had to find the solution with the fewest moves, it would be more difficult. Perhaps tradeoffs of this form between plan quality and the speed of planning could be incorporated into the Planning Graph Analysis paradigm.

Accessing Graphplan

Graphplan, including source code, sample domains, and several animations is available via http://www.cs.cmu.edu/~avrim/graphplan.html.

Acknowledgements

We thank Jaime Carbonell and the members of the CMU Prodigy group for their helpful advice.

References

[Barrett and Weld, 1994] A. Barrett and D. Weld. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

[Blum and Furst, 1995] A. Blum and M. Furst. Fast planning through planning graph analysis. In *IJCAI95*, pages 1636–1642, Montreal, 1995.

[Bylander, 1994] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[Carbonell, personal communication] J. Carbonell. 1994. personal communication.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[Etzioni, 1990] O. Etzioni. *A Structural theory of explanation-based learning.* PhD thesis, CMU, December 1990. CMU-CS-90-185.

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

[Goldberg and Tarjan, 1986] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[Knoblock, 1994] C. Knoblock. Generating parallel execution plans with a partial-order planner. In *AIPS94*, pages 98–103, Chicago, 1994.

[McAllester and Rosenblitt, 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 634–639, July 1991.

[Srinivasan and Howe, 1995] R. Srinivasan and A. Howe. Comparison of methods for improving search efficiency in a partial-order planner. In *IJCAI95*, pages 1620–1626, Montreal, 1995.

[Stone *et al.*, 1994] P. Stone, M. Veloso, and J. Blythe. The need for different domain-independent heuristics. In *AIPS94*, pages 164–169, Chicago, 1994.

[Veloso and Blythe, 1994] M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. In *AIPS94*, pages 164–169, Chicago, 1994.

[Veloso, 1989] M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, Carnegie Mellon University, December 1989.

[Weld, 1994] D. Weld. An introduction to partial-order planning. *AI Magazine*, 1994.