Fast Planning Through Planning Graph Analysis*

Avnm L Blum
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891
avnm@cs emu edu

Merrick L Furst
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891
mxfocs cmu edu

Abstract

We introduce a new approach to planning in STRIPS-like domains based on constructing and analysing 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

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 (PGA) In this approach, a compact structure we call a *Planntng Graph* is explicitly created that is then explored in a search A Planning Graph is *not* a graph of "world-states" (which of course could be huge) Rather, Planning Graphs are closer in spirit to the PSGs of [Etzion, 1990] Planning Graphs are structures based on domain information, the goals and initial conditions of a problem, and an explicit notion of time Planning Graphs offer a convenient, efficient means of organizing and maintaining search information They do so in a way that is reminiscent of the efficient solutions to Single-Source Shortest-Paths and Dynamic Programming problems Planning Graph Analysis appears to have significant practical value in solving planning problems even though the inherent complexity of STRIPS-like planning, which is at least PSPACE-hard, *IS* much greater than the complexity of Shortest-Paths or standard Dynamic Programming problems

Graphplan combines aspects of both total-order and partial-order planners On the one hand, Graphplan makes more commitments than traditional total-order planners On the other hand, the plans it generates are partially-ordered plans

The way in which Graphplan "over-commits" is that 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 It may seem puzzling that an extra level of commitment would lead to a fast planner, especially given the success enjoyed by least-commitment planners [McAllester and Rosenblitt, 199l][Barrett and Weld, 1994][Weld, 1994] However, the extra level of commitment allows Graphplan to store and manipulate valuable search information This enables it to rapidly determine when backtracking is needed

Even though Graphplan makes strong commitments, the plans it generates are partially-ordered plans For instance, in the 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, 19941, since one could imagine executing the actions in three time steps jf one had as many workers as needed to load and unload and fly the rockets

One valuable feature of our algorithmic 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" [Russell, 1992], the plan produced by

"This research is sponsored IN put by the Wright Laboratory, Aeronantical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-M330 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

The rocket domain has three operators 'Load', 'Unload', and 'Move' In UCPOP format, they 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)))
```

The key point to notice is that to move a rocket it must have fuel, and once the move is completed, the fuel is exhausted 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 *The rdcket* domain

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, unhke 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

1 1 Definitions and Notation

Planning Graph Analysis applies to STRIPS-hke planning domains In these domains, operators do not create or destroy objects and time may be represented discretely because the operators all act as atomic actions

Specifically, by a *planning problem,* we mean

- A STRIPS 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-instontiated operator For instance, the operator 'put ?x Into *y' may instantiate to the specific action ⌜pnt 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 the PGA 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 In a valid plan several actions may be specified to occur at a single time step as long as none of them deletes a precondition or Add-Effect of another [1] In a linear plan these *independent* parallel actions could be arranged in any order with exactly the same outcome It is legal to perform an action at time 1 if its preconditions are all in the Initial Conditions It is legal to 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/ only if it is an Add-Effeet 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 be independent

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, etc

Edges in a Planning Graph explicitly represent relations between actions and propositions The action nodes in action-level are connected by "precondition-edges" to their preconditions in proposition level t, 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]

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

[3] A graph is called *leveled* if its nodes can be partitioned into disjoint tela L_1, L_2, , L_n inch 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* —> *b* in a partial-order planner

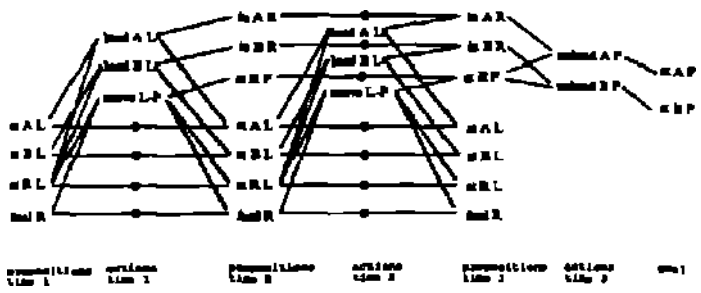

Figure 2 *A planning* graph for *the rocket problem with* one rociet A, *two* pieces of cargo A and B, a start location *L* and *one destination P For simplicity, the "rocket"* parameter *has been removed from the actions Delete edges* are *represented by dashed lines and no-ops* are represented *by dots*

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 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 1 (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 : +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 la 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 are two ways in which actions *a* and 6 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-Effeet 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 6 that axe 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 'Rocketl at London' in the Initial Conditions, the actions 'move Rocket 1 iron London to Paris' and 'load Alex into Rocketl in London' at time 1 are exclusive because the first deletes the proposition 'Rocketl at London' which is a precondition of the second The proposition 'Rocxetl at London' and the proposition 'Rocketl 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 Rocketl in London' and 'load Jason into Rocketl 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 'Rocketl at London' and "Rocketl at Paris'

Note that the Competing Needs notion and the exclusivity between propositions are not just logical properties of the operators They depend on the interplay between operators and the Initial Conditions For instance, the exclusivity of 'Rocketl at London' and 'Rocketl at Paris' cannot be logically concluded from the structure of the 'move' operator alone It is derived both from the structure of the operator and the fact that the rocket starts in only one place at the initial time

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 Rocketl at London (and they are nowhere else at time 1), then 'Alex in Rocketl' and 'Rocketl at Paris' are exclusive at time 2, but not at time 3

3 Description of the algorithm

The high-level description of our basic algorithm is the following Starting with a Planning Graph that only has

*In fact, determining *all* mutual exclusion relationships can be as hard as finding a legal plan For instance, consider creating two new artificals goals g1 and g_2 such that satisfying g_1 require! satisfying half of the original goals and satisfying g_2 requires satisfying the other half

a single proposition level containing nodes corresponding to Initial Conditions, Graphplan runs in stages In stage *i* Graphplan takes the length *i* — 1 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 (m which case it goes on to the next stage)

This basic algorithm could be termed "weakly complete" in each iteration through the Extend/Search loop described above, the algorithm either discovers a plan or else proves that no plan having that many time steps or fewer is possible In Section 5 we describe how this algorithm may be augmented to be complete in the usual sense, so that if the Problem Goals are not satisfiable by any valid plan then the planner does eventually halt with failure We wish to point out, however, that weak completeness can be useful For instance, a user may know that if the problem is solvable at all, then it will be solvable in, say, 15 time steps In that case, by repeating the Extend/Search loop until it has failed on a graph with 15 action levels, Graphplan is able to report failure in what may be significantly less time than would be needed to prove full unsolvability

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 the preconditions are labeled as mutually exclusive* [5] AIBO 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-or list for each action

To create a generic proposition level, simply look at all the Add-Effects of the actions in the previous level and place them in the next level as propositions At this point insert the add-edges and delete-edges (Note do not create a proposition if its only reason for existence is to be deleted by some action) 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 for problems with STRIPS-style operators

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 t 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_r I, and t*

Proof Let *k* be the largest number of formal parameters IS any operator Since operators cannot create new objects, the number of different propositions that can be created by instantiating an operator is $0(£n^k)$ So, the maximum number of nodes in any proposition-level of the planning graph is $0(p + mln')$ Since any operator can be instantiated in at most $0(n^k)$ distinct ways, the maximum number of nodes in any action-level of the planning graph is $0(mn^4)$ Thus the total size of the planning graph is polynomial in n, m, p, /, 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 B

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 (A) all the Problem Goals appear, and (B) 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 mapping these goals to some other set of goals at time *t* — 1 having the property that if only *these* goals could be achieved in (— 1 steps, then the original goals could be achieved in *t* steps If the goals at time *t* — 1 turn out to not be mutually solvable, Graphplan tries to find a different set of actions, yielding a different set of subgoals at time *t* — 1 and so forth, 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 method (easily implemented recursively) to generate the subgoal sets at time 1—1 from a given set of goals at time *t* For each goal at time *t*, for each action generating that goal (starting with the no-op) select that action if it is not exclusive of some action already selected Continue in this fashion with the next goal at time *t* and so forth If there are no actions available for

[5] Checking for exclusions keep* Graphplan, for instance, from inserting the action 'unload ilex from Rocket 1 In Pari** in time 2 of the rocket-domain graph when the initial condition! specify that both Alex and the rocket begin in London

achieving the current goal that are not exclusive of previous selections, then back up to the previous goal 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 Graphplan then continues this procedure at time step *t* - 1

An improvement to this approach (which is implemented in Graphplan and helps modestly on the tasks we have tried) 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

One final aspect of Graphplan'B search is that when a set of (sub)goals at some time i 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 i in a hash table Similarly, when it creates a set of subgoals at some time 1, 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

The strategy of working on the subgoals in a somewhat breadth-first-like manner makes Graphplan fairly insensitive to goal-orderings In particular, the number of sets examined did not depend much on the order of the goals in any of the domains we tried Within a Planning Graph level, the amount of time needed to construct a goal set at the previous time step from the current goal set might vary based on the ordering Nonetheless, empirically, Graphplan's dependence on goal orderings seems to be much less than that of other planners such as Prodigy and UCPOP

4 Experimental Results and Discussion

4 1 Natural domains

We compared Graphplan with two popular planners, Prodigy and UCPOP, on two "natural" planning problems For both problems we ran Prodigy with heuristics suggested in Stone et al [Stone *tt al*, 1994] and by Carbonell [Carbonell, personal communication] Note that Graphplan is written in C while the other planners are in compiled LISP On the other hand, we ran Graphplan on a (slow) DECstation 2100 and the other planners on a (faster) SPARC10

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 *i* — 1 from a goal set at time *t)* and the total number of recursive calls made (the number of times it selects an action) These are somewhat analogous to the back ward-chaining steps taken by total-order planners

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 [n/2] of the objects to Paris and *[n/2j* of the objects to JFK The goals are ordered alternating between destinations

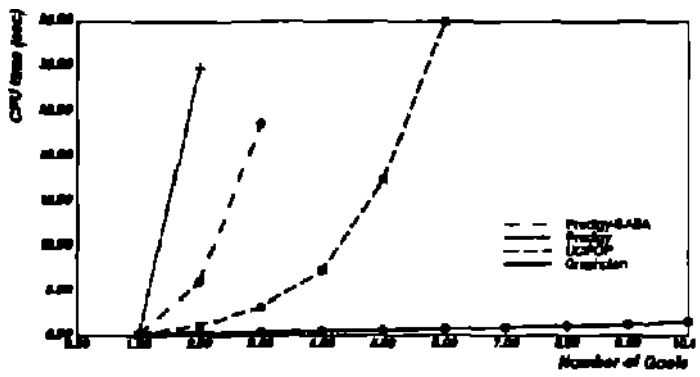

Figure 3 *2-Rockets problem*

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 Rocketl 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 recursive calls is linear in the number of goals

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

Flat Tire

A "natural" problem of a different sort is Stuart Russell's "fixing a flat tire" domain [Russell, 1992] 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 *bom* a minimum of 107 to a maximum of 246, and the number of recursive calls from 609 to 1380 Neither UCPOP nor Prodigy found a solution 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 it also produces a "sensible" plan Figure 4 shows the plan produced by Graphplan for this problem

4 2 Artificial domains

The papers by [Barrett and Weld, 1994] and [Veloso and Blythe, 1994] define a collection of artificial domains intended to distinguish the performance characteristics of

"Graphplan currently makes no attempt to order the goals at a tune step in an advantageous way We are currently experimenting with various standard heuristics

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*

various planners On all of these, Graph plan is quite competitive with the beet 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

4 3 Discussion of Experimental Results

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

Mutual Exclusion The propagation and extensive use of mutual exclusion relations 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 places in time, Graphplan is able to record the goal seta 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

It is interesting to note that in three out of four of these points Graphplan's commitment to putting specific actions at specific points in time plays an important role

5 Making Graphplan Complete

To a first approximation, Graphplan conducts something similar to an iteratively-deepened search In the i^{th} stage

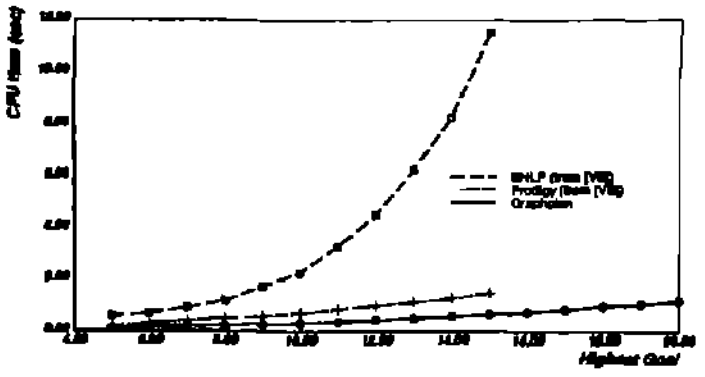

Figure 5 *Link-repeat domain from (Veloso & Blythe 1994)*

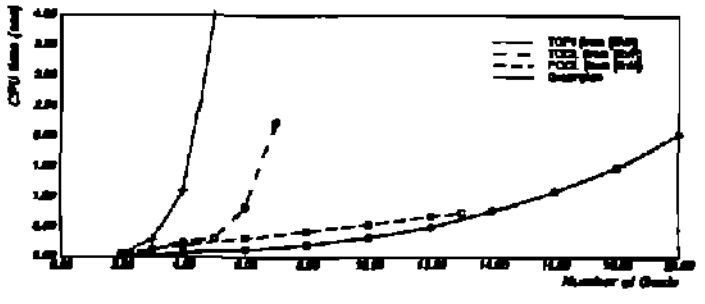

Figure 6 D^1S^1 *domain from (Barrett & Weld 1994)*

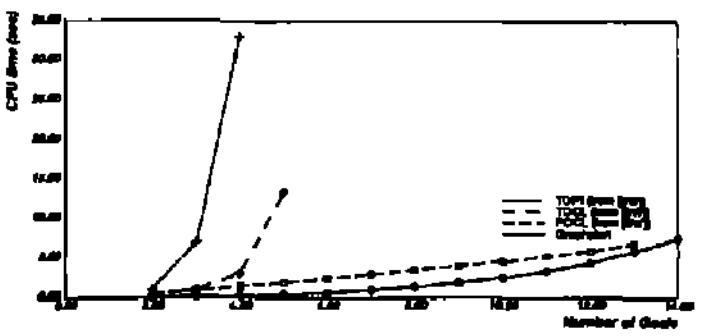

Figure 7 D^1S^2 *domain from (Barrett & Weld 1994)*

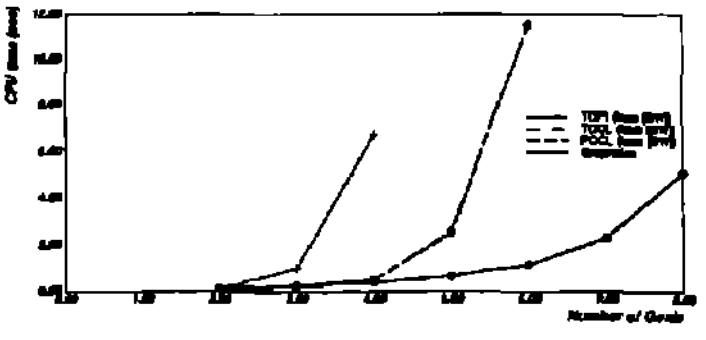

Figure 8 D^mS^{2*} *domain from (Barrett & Weld 1994)*

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 that makes Griphplan a *complete* planner That is, by augmenting Graphplan with the test we will describe, if the input problem has DO solution then Graphplan will eventually halt and say "No Plan Exists[n]

6 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's, 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 Bet 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 The Test that Makes Graphplan Complete

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 However, it may be the case that no plan exists but this simple test does not detect it, so we need to do something slightly more sophisticated

As mentioned earlier, Graphplan memoizes, or records, goal sets that it has considered at some level and determined to be unsolvable Let *S1* be the collection of all such sets stored for level t after an unsuccessful stage *t* In other words, after an unsuccessful stage *t,* Graphplan has determined two things (1) any plan of / or fewer steps must make one of the goal sets in *S1* true at time i, and (2) none of the goal sets in *S1* are achievable m i steps The modification to Graphplan to make it complete is now just the following

> If the graph has leveled off at some *]evel n* and a stage 1 has passed in which $\|S^t_n\sim^1\| = |SJ|$, then output "No Plan Exists "

Theorem 2 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^t_n is never *smaller* than the number of sets in S^{t-1}_n, 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^t_{n+1} must, one step earlier, achieve some set in S^t_n This is because of the way Graphplan works it determined each set in S^t_{n+1} 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, $S^t_{n+1} = S^{t-1}_n$ 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, $|S^{t-1}_n| = |S^t_n|$ (which implies that $S^{t-1}_n = S^t_n$) This means that $S^t_{n+1} = S^t_n$ Thus, in order to achieve any set in S^t_{n+1} 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 ■

Graphplan 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

[Carbonell, personal communication] J Carbonell 1994

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

[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

[Russell, 1992] S Russell Efficient memory bounded search algorithms In *ECAI 92*, Vienna, 1992

[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

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