Understanding and Extending Graphplan

Subbarao Kambhampati, Eric Parker and Eric Lambrecht †
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
http://rakaposhi.eas.asu.edu/yochan.html; {rao,ericpark,eml}@asu.edu

Abstract

We provide a reconstruction of Blum and Furst's Graphplan algorithm, and use the reconstruction to extend and improve the original algorithm in several ways. In our reconstruction, the process of growing the planning-graph and inferring mutex relations corresponds to doing forward state-space refinement over disjunctively represented plans. The backward search phase of Graphplan corresponds to solving a binary dynamic constraint satisfaction problem. Our reconstruction sheds light on the sources of strength of Graphplan. We also use the reconstruction to explain how Graphplan can be made goal-directed, how it can be extended to handle actions with conditional effects, and how backward state-space refinement can be generalized to apply to disjunctive plans. Finally, we discuss how the backward search phase of Graphplan can be improved by applying techniques from CSP literature, and by teasing apart planning and scheduling (resource allocation) phases in Graphplan.

1. Introduction

Blum and Furst's Graphplan algorithm [1995] has recently emerged as the fastest planner for solving classical planning problems. Despite its impact, currently there exists very little critical analysis of the algorithm, its sources of strength and its relation to the traditional planning algorithms. In this paper, we will show that significant surface differences notwithstanding, the Graphplan algorithm can actually be best understood in terms of forward state-space refinement over disjunctive partial plans that correspond to a unioned representation of the forward state-space search tree. Normally, direct refinement of these disjunctive plans (called "planning-graphs" in Graphplan parlance) tends to result in reduced "pruning power." To counter this, Graphplan infers and propagates information about disjuncts that cannot together hold in any solution (called "mutex" relations in Graphplan parlance). The mutex inference process can be seen as a generalization of the standard refinement strategies in the presence of disjunctive plans, and helps Graphplan in producing a planning-graph that is a close "upper bound" approximation to the forward state-space search tree. Solution extraction involves searching for a sequence of actions that is part of the disjunctive plan, and solves the planning problem. This can be done by a backward search over the planning-graph structure. The search problem here corresponds closely to a binary dynamic constraint satisfaction problem [Mittal & Falkenhainer, 1990].

Our reinterpretation of Graphplan, summarized above, explains the role of such properties as "interference" relations, "mutex" relations and "action parallelism" in the algorithm, and also clarifies their relative impact on its performance. Our reconstruction also helps us place Graphplan in the pantheon of refinement planners. The connection between Graphplan and forward state-space refinement suggests improving Graphplan by making it goal-directed, and extending it to handle more expressive actions. The connec-

† This research is supported in part by the NSF NYI award IRI-9457634, the ARPI Initiative grant F30602-95-C-0247 and the ARPA AASERT grant DAAH04-96-1-0231. We would like to thank Avrim Blum and Dan Weld for helpful discussions on Graphplan, and Mark Peot and David Smith for making their Lisp implementation of Graphplan available to us.

tion between the backward search on the planning-graph and the dynamic constraint satisfaction problem suggests improvements to backward search by exploiting techniques from constraint satisfaction (c.f. [Tsang, 1993]).

The rest of the paper presents our reconstruction of the Graphplan algorithm, and elaborates on the insights it affords and the extensions it suggests. The paper is organized as follows: In Section 2, we will reconstruct the Graphplan algorithm starting from a generalized forward state-space search. In Section 3, we discuss the insights offered by our reconstruction. Section 4 presents a variety of ways of improving the efficiency and coverage of the Graphplan algorithm, including making Graphplan goal directed through means-ends analysis (4.1), supporting propagation of mutex information in the backward direction (4.2), empowering Graphplan to handle more expressive action representations (4.3), improving Graphplan's backward search phase with the help of CSP techniques (4.4), and teasing apart planning and scheduling in Graphplan (4.5). Section 5 summarizes the contributions of this paper.

2 Reconstructing Graphplan from Forward State-space Search

2.1 Generalized Forward State-space search

Let us start our reconstruction of Graphplan with a simple example domain containing 6 operators -- $O_1, O_2, \ldots O_6$, and a No-OP (do nothing action), all of whose preconditions and effects are shown on the top in Figure 1. Now consider the process of forward state-space search starting from the initial state (R,W). Recall that this involves applying every applicable action to the initial state to compute new states, and applying operators to the resultant states iteratively. A portion of the search tree resulting from this process is shown on the bottom left in Figure 1 (in solid lines). Each path in this tree corresponds to a partial plan. Planning ends when one of the states in the search tree contains all the goals. The path to that state is output as the plan.

Although this process normally leads to "sequential" or totally ordered plans, we can easily extend the idea to allow for a special form of parallelism in the solutions. This is done by projecting the sets of actions that are independent of each other (i.e., they don't delete each others preconditions or added effects) together [Drummond, 1989] [1]. For the search tree on the bottom left in Figure 1, the portion of the tree produced by allowing the projection of independent action sets is shown in dashed lines. For example, since O_1 and O_2 are independent and have their preconditions satisfied in the initial state, we can apply the action set (O_1,O_2) together to the state (R,W) to get the state (P,Q) at the second level, from where we can apply O_4 to get Z in the third level. If our goal is Z, we can terminate in the third level with the solution $[O_1,O_2]$--O_4. This plan is said to be two time steps long, with the first time step containing two actions. The semantics of the parallelism is that in the first time step, we can execute O_1 and O_2 concurrently, and in the second step we execute O_4. It should be easy to see that as long as we allow the projection of all independent action sets, any complete search strategy (such as breadth first) is guaranteed to find us the shortest plan (in terms of number of time steps). If we only project single actions, the first time Z appears in a state is at the fourth level. In other words, allowing the projection of independent action sets reduces the solution depth (while increasing the branching and redundancy in the search tree). The total number of actions in the solution however remain the same.[2]

---

[1] This is however very different from the kind of partial order found in plan space plans. In particular, within a given branch of the projection tree, the positioning of the operators with respect to time steps is strictly fixed; this positioning is what gives the state information at the next level.

[2] It is possible to reduce the redundancy in this search tree by first pre-processing the domain to partition actions into maximal sets of non-interfering actions, and then doing search by projecting each of the action sets, rather than individual actions. Since not all actions in an action set may be simultaneously applicable in a given state, projection will involve stripping the action set of inap-

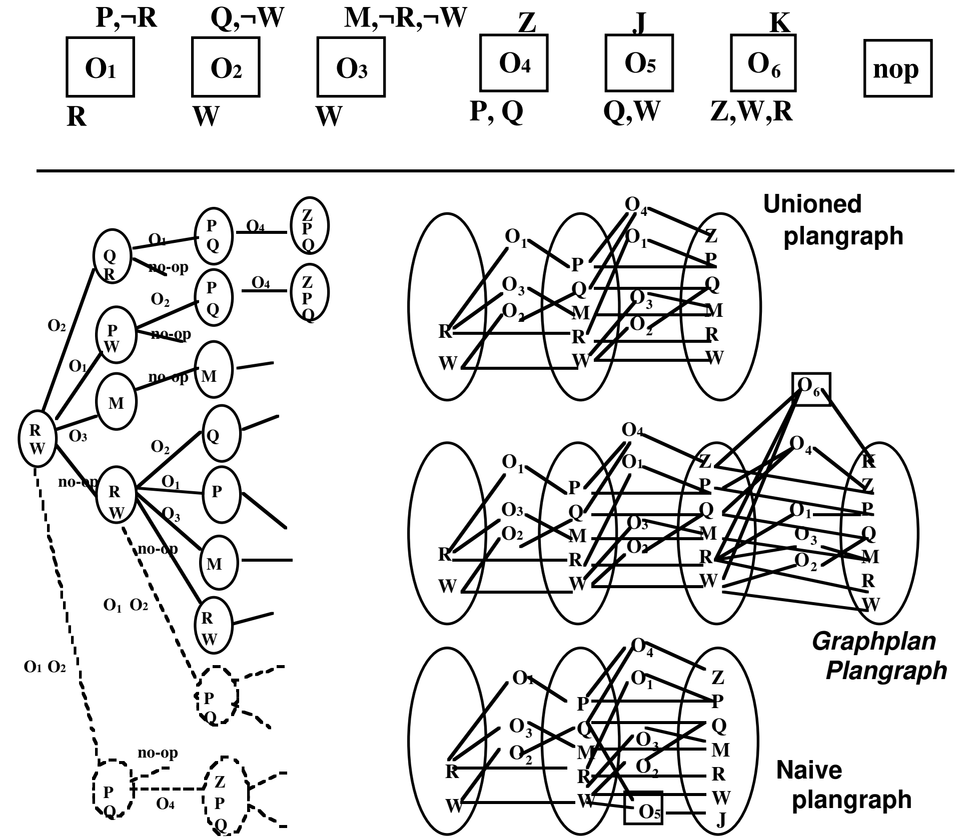

Figure 1. Interpreting Graphplan as a disjunctive forward state-space refinement planner. On the left is a portion of the search tree generated by a forward state-space search. The portion in dashed lines show the branches produced by allowing the parallel projection of independent actions. On the right are three different disjunctive representations of the tree.

2.2 Planning-graph as a compact representation for the search tree

A big problem with forward state-space search (that it shares with many other planning algorithms), is that the search (projection) tree can grow exponentially large even for very small problems. One way of getting a handle on the forward branching would be to somehow consider all the states in a given level *together*, rather than as separate nodes in the search tree. In particular, seeing the states as sets of propositions, we can try to union the sets together to get the collection of propositions from which all the states in that level are made up. These unions are roughly (see below) equivalent to what the Graphplan algorithm calls "proposition lists." We shall call the resulting structure a "*unioned planning-graph"*.

The unioned planning-graph for our example is shown at the top right in Figure 1. The proposition lists at various levels are shown in ovals. The proposition list at the first

plicable actions. Finally, since not all the actions in an action set may not be relevant to the final goal, the sequence of action sets that is returned upon termination will have to be post-processed to eliminate irrelevant actions. This can be done in polynomial time with the help of the causal structure of the solution plan.

level is the union of just the initial state, and is thus (R,W). At the second level, it is (P,Q,M,W,R), and at the third level, it is (Z, P, Q, M, W, R).

We can relate the proposition lists at successive levels in terms of actions that can be taken between these levels. Specifically, the unioned planning-graph in Figure 1 contains, between every two consecutive proposition lists, all the actions taken between those levels in the corresponding state-space projection tree shown in Figure 1. The actions in the planning-graph can be connected to the elements in the preceding proposition list that comprise their preconditions, and the elements in the following proposition list that comprise their effects. Since the actions are only connected to the propositions they change, the persistence of propositions during action execution is not explicit in the planning-graph. To make the persistences explicit, a set of "no-op" actions -- one for the persistence of each of the propositions -- are introduced. In the unioned planning-graph in Figure 1, these correspond to the direct connections between the propositions in consecutive levels.

Loosely speaking, the unioned proposition list at level **i** in the planning-graph can be seen as a compact "disjunctive" representation of the $\mathbf{b}^{\mathbf{i}}$ states (assuming a branching factor b) at the $\mathbf{i}^{th}$ level of the corresponding state-space search tree. Thus a unioned planning-graph does seem to provide a compact representation for curtailing the growth of forward state-space search. Before we can make this representation the basis for an efficient planning algorithm, we need to decide how the planning-graph structure is grown, and how solutions are extracted from it.

2.3 Growing and searching planning-graphs

Although the unioned planning-graph itself can be grown by generating and converting the state-space search tree, this is clearly very inefficient as it will have the overhead of classical state-space planning. We can reduce the cost significantly if we are willing to get by with a planning-graph structure that is an (upper bound) approximation to the unioned planning-graph.

Specifically, one simple way of growing a planning-graph would be to add all actions (including no-ops) whose preconditions are subsets of the current proposition list, and construct the proposition list at the next level as the union of the effects of all these actions.[3] We shall call the resultant structure the "*naive planning-graph.*" The naive planning-graph for our example problem is shown in Figure 1. Note that O_5 (in the second level of the naive planning-graph) is not present in the state-space search tree in Figure 1 (and thus also absent from the unioned planning-graph in Figure 1), making the naive planning-graph an upper bound approximation to the unioned planning-graph. In contrast to the exponential cost of generating the state-space search tree, the naive planning-graph is a polynomial-sized structure that can be generated in polynomial time (the main savings in generation cost comes because the planning-graph contains at most one instance of each action at each level, as against the possibly exponential number in the corresponding level of the state-space search tree).

To do solution extraction, we need to prove to ourselves that there is a sequence of legal states culminating in a state where all the goals are satisfied. This can be done by essentially following the dependency links between actions and proposition list elements. For example, to see if we have a successful plan for goals (P, Z) in the planning-graph shown in Figure 1, we check if there are actions supporting P and Z in that level. Z is supported by O_4 while P is supported by O_1 and no-op. Since we need only one support for each goal, we consider each support of P in turn. Suppose we pick O_1 for P (leaving no-op as a backtrack choice) and O_4 for Z. We then have to ensure that the preconditions of these two actions are satisfied in the previous proposition list. This leads to the recursive goal set (R, P, Q) at the previous level. At this point, we have single supports to all three goals—no-op for R, O_1 for P and O_2 for Q. These actions cannot occur together

[3] An even simpler way would be to introduce all actions at all levels. This is the idea used by Kautz and Selman [1996] in their linear encodings.

since R is deleted by O_1. We can think of this as O_1 interfering with no-op. At this point we can backtrack, consider the remaining choice, no-op, as the support for P, and we will be able to succeed.

The termination process discussed above can be cast as a ***dynamic constraint satisfaction problem (DCSP)*** [Mittal & Falkenhainer, 1990]. The DCSP is a generalization of the Constraint Satisfaction Problem [Tsang, 1993], that is specified by a set of variables, activity flags for the variables, the domains of the variables, and the constraints on the legal variable-value combinations. In a DCSP, initially only a subset of the variables is active, and the objective is to find assignments for all active variables that is consistent with the constraints among those variables. In addition, the DCSP specification also contains a set of "activity constraints." An activity constraint is of the form: "if variable **x** takes on the value $\mathbf{v_x}$, then the variables **y,z,w...** become active." The correspondence between the planning-graph and the DCSP should now be clear. Specifically, the propositions at various levels correspond to the DCSP variables, and the actions supporting them correspond to the DCSP domains. The action interference relations can be seen as DCSP constraints -- if $\mathbf{a_1}$ and $\mathbf{a_2}$ are interfering with each other, and $\mathbf{p_{11}}$ and $\mathbf{p_{12}}$ are supported by $\mathbf{a_1}$ in the planning-graph while $\mathbf{p_{21}}$ and $\mathbf{p_{22}}$ are supported by $\mathbf{a_2}$, then we have the constraints: $\mathbf{p_{11}=a_1} \Rightarrow \mathbf{p_{21} \neq a_2}$, $\mathbf{p_{12}=a_1} \Rightarrow \mathbf{p_{21} \neq a_2}$, etc. Activity constraints are implicitly specified by action preconditions: supporting an active proposition **p** with an action **a** makes all the propositions in the previous level corresponding to the preconditions of **a** active. Finally, only the propositions corresponding to the goals of the problem are "active" in the beginning.

2.4 Improving growth and search of planning-graphs

The naive planning-graph can be exponentially larger (in terms of actions per level) than the unioned planning-graph (to see this, consider a situation where the preconditions of all domain actions hold in a proposition list, but only a few subsets of the proposition list actually correspond to legal states). Thus, growing and searching it may actually be worse than doing state-space search.

Thankfully, we can improve our approximation to the unioned planning-graph by tracking information as to which subsets of the proposition list do not belong to legal states. For example, if we recognize that (Q,W) is not part of any legal state at the second level of the search tree in Figure 1, we could avoid projecting the action O_5 (which has preconditions Q,W) from this proposition list. This type of information can also control the backward search for a solution on the planning-graph. For example, if we know that (R, P, Q) is not a part of any legal state at the second level, we could backtrack as soon as we get this as our recursive goal set.

Unfortunately, keeping track of all illegal subsets of a proposition list can in general require both too much computation (since we don't have access to the state-space search tree) and memory (as there can be an exponential number of illegal subsets of a proposition list in the worst case). However, even knowing *some* of the illegal subsets of the proposition lists can help us. An interesting middle ground is to keep track of all 2-sized illegal subsets alone (since there are only at most n^2 such subsets). This is the idea behind Graphplan's "mutex" relations—two elements in a proposition list are said to be mutex if they belong to the set of 2-sized illegal subsets.

The nice thing about pair-wise mutex relations is that they can be inferred efficiently through incremental constraint propagation. We start by generalizing the mutex relations to hold between pairs of interfering actions. Although the Graphplan algorithm assumes a specific definition of interference -- viz., actions interfere if the effects of one violate the preconditions or effects of the other, the definition of interference between actions is really up to the designer of the domain, and only needs to capture the requirements for concurrent execution [Reisig, 1982].[4] Starting with the actions in the first level, we can

---

[4] It is instructive to note that optimal solution length depends on the interference definition used. For example, if the domain doesn't allow any concurrency, we can consider every pair of actions to be interfering. In this case, optimal solutions will be sequential. Similarly, we can weaken the

consider any pair of actions that interfere to be mutex. The mutex relation can then be propagated to any effects of the actions in the next proposition list (as long as those effects are not being supported by other non-interfering actions). In particular, in the example we have been following, O_1 and the no-op leading to R are interfering as the former deletes R. Thus, R and P are mutex at the second level proposition list. Similar reasoning shows that Q and W are mutex.

As expected, mutex information curtails the growth of the planning-graph and helps in termination check. Since (Q,W) are found to be mutex, there is no point in projecting O5 from the first level proposition list, as is shown in the Graphplan planning-graph in Figure 1 (mutex relations are not shown in the figure). Similarly, since R and P are mutex at the second level, the backward search for the goals P and Z can be terminated as soon as we reach the second level proposition list.

In the context of our observation in the previous section that the planning-graph can be seen as a DCSP, the mutex propagation can be seen as inferring additional variable-value constraints. The mutex propagation process however is different from the standard notion of "constraint propagation" in the CSP. To see this, note that the mutex propagation routines use the constraints among variables (propositions) at one level to derive implicit constraints among the variables at the next level. In contrast, normal constraint propagation procedures use the constraints among a given set of variables to derive new constraints among the same set of variables.

3 Discussion

The reconstruction of the Graphplan algorithm in terms of forward state-space projection reduces the mystery regarding the antecedents of the algorithm. We recognize that both the fact that plans produced by Graphplan can contain multiple actions within each step, and the fact that Graphplan terminates with smallest plan (in terms of time-steps), are consequences of the properties of the corresponding forward state-space search tree. Similarly, we recognize that the planning-graph is an *approximation* to the disjunctive (unioned) representation of the state-space search tree—with the approximation becoming finer and finer as we maintain bookkeeping information about illegal subsets of proposition lists. We have shown that mutex relations can be understood as inferring information about 2-sized illegal subsets.

Our reconstruction shows that the ability to refine disjunctive partial plans, rather than action parallelism, is the more important innovation of the Graphplan algorithm. In particular, it is easy to make Graphplan produce only serial plans by changing our definition of action interference, and considering every pair of non no-op actions to be interfering. This serial version of Graphplan will have the same tradeoffs with respect to Graphplan as normal forward state-space search will have with search with projection of independent action sets. Specifically, in situations where the domain does allow parallelism, the parallel planning-graph can terminate with fewer levels than the serial planning-graph. More importantly, the serial version of Graphplan will still outperform normal state-space planners. We verified this observation by comparing a state-space planner based on Graphplan data-structures to a serial version of Graphplan.

Our account also gives us some insights into the tradeoffs offered by the propagation of mutex relations. The first point we note is that mutex relations help in generating a planning-graph that is a finer approximation to the unioned planning-graph. The approximation to the unioned planning-graph provided by the mutex relations will in general differ from the exact unioned planning-graph since mutex identifies only 2-sized illegal subsets but not the 3- and higher-sized ones. Because of this difference, some illegal projections are still left in the planning-graph. In our example in Figure 1, the Graph-

---

interference to hold only between actions one of which deletes the preconditions or *useful* effects of the other (notice that Graphplan considers violation of any effect, not just useful effects). This weaker interference relation will lead to more parallel (and thus shorter) solutions than Graphplan.

plan planning-graph includes O_6, with preconditions Z, R, and W, in the third level actions, and the proposition K in the level four propositions, although it is easy to see from the search tree that O_6 cannot be applied from any of the states in level 3.

While the approximation can be further improved by 3- and higher sized mutex relations, whether or not the cost of computing the mutex relations is offset by the improvements in planning-graph size and the time to search it depends on what percentage of action interactions in the domain are pair-wise (as against three-, four-, or n-ary interactions).[5] The amazing practical success of 2-sized mutex relations can be explained by the relative rarity of higer-order interactions between actions in most classical planning domains. This rarity is to a certain extent due to a lack of global resource constraints in the classical planning domains. 2-sized mutexes may thus not be as effective when we consider more complex domains containing such global constraints. To illustrate, consider a variation of the blocksworld domain where the blocks have non-uniform sizes, with super-blocks that can, for example, hold two blocks on top of them. In such a scenario, suppose we are considering three actions all of which attempt to stack a distinct block on top of the same super-block. These three actions together are interacting, while no two of them are interacting.

The connection between the planning-graph and the dynamic constraint satisfaction problem provides insights into the operation of the backward search phase of the Graphplan algorithm. In essence, the Graphplan algorithm uses a systematic backtracking search strategy [Tsang, 1993] for solving the dynamic CSP—computing a satisfying assignment at one level, and recursing on the variables that are activated in the next level. Upon failure to compute a satisfying assignment at any level, the algorithm backtracks to the previous level. The memoization phase in Graphplan can be seen as a special form of no-good learning [Kambhampati, 1997b]. The connection also makes it clear that the specific details of the backward search algorithm used in Graphplan are by no means unique. There exist other possible ways of solving dynamic CSP algorithms that are worth considering. These include variable and value ordering heuristics as well as full no-good based learning schemes. We shall elaborate these in Section 4.4.

3.1 Graphplan as a refinement planner

Most existing AI planners fall under the rubric of refinement planners. Kambhampati et. al [1995,1996,1997c] provide a general model for refinement planning. Our reconstruction of Graphplan from forward state-space planning helps us interpret it as an instance of this generalized refinement planning. Specifically, the planning-graph structure of Graphplan can be seen as a "partial plan" representation. An action sequence belongs to the candidate set of the planning-graph if it has a prefix that contains some subset of actions from each of the levels of the planning-graph, contiguous to each other. For example, any action sequence with a prefix O_1-O_2-O_4-O_6 will be a candidate of the planning-graph shown in Figure 1 (since O_1, O_2 belong to the first level actions, O_4 belongs to the second level actions and O_6 belongs to the third level). O_1-O_2-O_4-O_6 itself is one of the minimal candidates of the planning-graph. Similarly, no action sequence that has a prefix O_6 can be a candidate of this planning-graph. The relation between the naive, unioned and Graphplan planning-graphs can be formally stated by saying that the candidate set of the naive planning graph is a superset of the candidate set of the Graphplan planning-graph, which in turn is a superset of the candidate set of unioned planning-graph.

The forward "planning-graph growing" phase corresponds to the refinement operation. The inference and propagation of mutex constraints can be seen as a legitimate part of the

[5] Even binary interactions between actions can eventually lead to higher-level interactions between propositions. Graphplan algorithm does remember higher level mutex relations between propositions by caching ("memoizing") failing goal sets. Graphplan algorithm works because these higher order relations are a small subset of the total number, and can be learned upon failure. In a domain with a significant number of non-binary action interactions, higher order interactions between propositions will be much more prevalent and cannot be postponed to the learning stage without a drastic hit on performance.

refinement process since the addition of mutex constraints reduces the candidate-set of the planning-graph. The backward search phase of Graphplan corresponds to “solution extraction”—to see if any of the minimal candidates of the planning-graph correspond to a solution. It is thus easy to see that solution extraction is helped by the reduction in the candidate-set of the planning-graph that is achieved through the inference of mutex constraints.

The interesting thing about this analogy is that unlike traditional refinement planners, such as the forward state-space planner or the partial-order planner, which search among various partial plans during the refinement phase, Graphplan considers only one partial plan and thus does not incur *any* premature commitment. As we saw, this single plan is approximately equal to a unioned or “disjunctive” representation of all the partial plans in the forward state-space search. While there is no search among partial plans in Graphplan, its solution extraction process is costlier than that used by a forward state-space refinement.

All this brings up a very important insight: If Graphplan can do so well by refining disjunctive plans using forward state-space refinements, might it not be possible to duplicate this success in the context of backward state-space, plan-space and task-reduction refinements? Kambhampati [1997a] considers this question and outlines the challenges involved in developing such planners.

4 Extending Graphplan

4.1 Making Graphplan goal-directed

The strong connection between forward state-space refinement and Graphplan suggests that Graphplan can suffer some of the same ills as the former. Specifically, in many realistic domains, there will be too many actions that are applicable in any proposition list, unduly increasing the width of the planning-graph structure. Indeed, we (as well as other researchers) found that despite its efficiency, Graphplan can easily fail in domains where many actions are available and only few are relevant to the top level goals of the problem. There are several ways of making Graphplan goal-directed.

One simple way of reducing the width of the planning-graph would involve constructing and using *operator graphs* for the problem. Operator graphs [Smith & Peot, 1996] symbolically back-chain on the goals of the problem to isolate actions of the domain that are potentially relevant for solving the problem. Although not every action in the graph is guaranteed to be relevant for the solution, we can guarantee that all solutions consist only of operators from the operator graph. Thus, restricting attention to these actions in the planning-graph expansion phase will reduce the width of the planning-graph. The cost of operator graph construction is itself quite low since the graph contains at most one instance of each precondition and action.

The operator graphs may still not be able to reduce the width of the graph sufficiently however. In particular, the fact that an action is relevant to the problem doesn’t imply that it is to be considered in each action level of the planning-graph. In order to predict the relevance of an action to a given level, we have developed a novel way by adapting means-ends analysis [McDermott, 1996] to Graphplan. Our approach, called Mea-Graphplan [Parker & Kambhampati, 1997]., involves first growing the planning-graph in the backward direction by regressing goals over actions, and then using the resulting structure as a guidance for the standard Graphplan algorithm. Growing the backward planning-graph involves regressing proposition lists over “relevant” actions. In particular, for every proposition p in the k^{th} level proposition list (counting from the goal state side), we introduce all actions (include the dummy *“preserve-p”* action) that have p in their effects list into the $(k+1)^{th}$ level action list. Actions are introduced only if they are not already present at that level. The proposition list at the $(k+1)^{th}$ level then consists of the preconditions of all the actions introduced at the $(k+1)^{th}$ level. Suppose we have grown the planning-graph in this way for j levels, and would like to search for the solution in the graph. The j-level backward planning-graph structure shows all actions that are relevant at each level of the forward planning-graph. We can now run the standard Graphplan algorithm, making it consider only those actions that are present at the corresponding level of the backward planning-graph. Alternately, we can also mark the propositions in the j^{th} level that are present in the initial state as "*in*" and propagate the "*in*" flags to actions and propositions at the earlier levels (an action is in if all its preconditions are in, and a proposition is in if any of its supporting actions are in). If this process results in the goal propositions being marked in, we can commence a mutex propagation phase on the part of the graph that is marked "*in*", followed by a backward search phase.

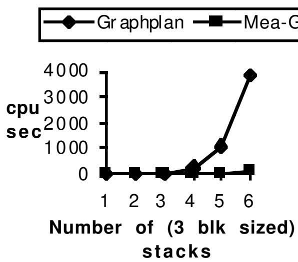

Figure 2. Performance improvement achieved by making Graphplan goal directed

Notice that the planning-graph considered by either of these approaches is significantly smaller than both the pure backward planning-graph (since some of the actions may not actually have their preconditions satisfied in the preceding action level) and the standard planning-graph (since we only consider actions that are relevant to the goals). This leads to a dramatic reduction of plan-graph building cost in problems containing many irrelevant actions. Figure 2 compares Mea-Graphplan with the standard Graphplan algorithm on a set of blocks world problems. Each problem consists of some k 3-block stacks in the initial state, which are to be individually inverted in the goal state. The graph plots the cpu time as k is increased (the planners were written in Lisp and run on a Sparc 10). Notice the dramatic increase in the cost of planning suffered by Graphplan as compared to Mea-Graphplan. The main reason for this turns out to be the graph build time. For values of k greater than 6, Mea-Graphplan still solves the problem while Graphplan runs out of memory.

4.2 Propagating mutexes in the backward direction

Although Mea-Graphplan generates the planning-graphs in the backward direction, it ultimately uses the forward state-space refinement ideas to infer and propagate mutex relations. A variation on this theme would involve doing the inference and propagation in the backward direction also, thus developing a Graphplan algorithm based completely on backward state-space refinement. Although it would look at first glance that no useful information can be propagated in the backward direction, we have found that the information about "goals that will never be required together" can be gainfully propagated.

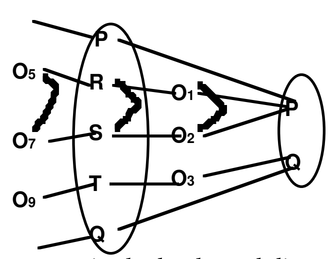

Figure 3. Propagating mutexes in the backward direction (mutexes shown as black links)

Specifically, two actions are said to be backward mutex if (a) they are statically interfering, i.e., the effects of one violate the preconditions or "useful" effects (the effects of the action that are used to support propositions in the planning graph) of the other, or (b) they have the exact same set of useful effects, or (c) the propositions supported by one action are pair-wise mutex with the propositions supported by the other. Two propositions are backward mutex if all the actions supported by one are pair-wise mutex with all the actions supported by the other. To illustrate this, consider the example shown in Figure 3. Here, O_1 and O_2 are mutex since they are solely supporting P. This leads to a mutex relation between R and S (since they are supporting actions that are mutex). Finally, O_5 and O_7 are mutex since they are supporting propositions that are mutex. Our preliminary studies[6] show that these propagation rules derive a reasonably large set of mutex relations on the planning-graph. We are investigating their utility in controlling search both by themselves and in concert with forward mutexes. We are also considering the effectiveness of the backward mutexes in controlling the search of the planning-graph in the forward direction (which corresponds to a variant of the network flow problem).

4.3 Handling more expressive action representations

Although the Graphplan algorithm is originally described only for propositional actions, there is nothing in the algorithm that inhibits it from being extended to more "expressive" action representations such as the ones used in UCPOP [Penberthy & Weld, 1992]. The UCPOP representation allows negated preconditions and goals, conditional and quantified effects in the actions, as well as disjunctive preconditions. We will briefly discuss how each of these can be handled.

Negated preconditions and goals are quite straightforward to handle, if we allow proposition lists to contain both positive and negative literals. Disjunctive preconditions can be handled by allowing actions at the k^{th} level to be connected to multiple sets of preconditions at $(k\text{-}1)^{th}$ level. Finally, universally quantified effects can essentially be expanded into a conjunction of unquantified conditional effects. Notice that this expansion does not increase the number of actions.

The treatment of conditional effects is more interesting. One obvious approach would be to convert each action with conditional effects into a set of actions *without* conditional effects. However, this approach often leads to an exponential blow-up in the number of actions. A better approach is to handle conditional effects directly. Doing this requires several minor modifications to the Graphplan algorithm. We shall illustrate these with the help of the example in Figure 4. Here, there are five actions, O_0 , ..., O_4. The actions O_3 and O_4 have conditional effects, as shown on the left-hand side of the figure. Suppose that we are starting from the initial state shown at the left-most proposition list of the planning graph and are interested in achieving goals *K* and *M*. The planning-graph will be grown as shown in the figure (to avoid clutter, we only show the persistence actions that will be useful in solving the problem, and avoid showing O_3 in the first level). The first iteration is straightforward with one small exception--we explicitly introduce negated propositions, and support persistence of negated propositions from the initial state by using the closed world assumption (e.g. ¬J, which is true in the initial state by closed world assumption, persists to the second proposition list). The actions O_1 and O_2 are interfering and thus the propositions (P, W) and (¬R, W) are marked as mutex. The extension at the second level involves actions with conditional effects. Propositions corresponding to conditional effects are handled with conditional establishment links. For example, L is provided by O_3 if P is true in the previous state. Thus we support L with a link to O_3 and its antecedent proposition P in the previous level. Unconditional establishment can be seen as a special case where the antecedent link is omitted (to signify that no special antecedents are required); see the effect K of O_3.

Next, we need to generalize the mutex propagation to work in the presence of conditional establishments. We do this as follows: two (conditional) establishment paths are

[6] This is part of joint work with Dan Weld.

mutex if (a) the action supporting one is mutex with the action supporting the other, or (b) one of the antecedents of the first conditional establishment is mutex with one of the antecedents of the second one, or (c) one of the antecedents of the first conditional establishment is mutex with the consequent of the second conditional establishment (or vice versa). The clauses "b" and "c" of course are the generalizations needed to handle conditional effects. Two propositions are mutex if each of the establishments of the first proposition is mutex with each of the establishments of the second proposition.

Applying this rule, we find that L and M are mutex at the third level since P, which is an antecedent of L, is mutex with W, which is an antecedent of M. Notice that O_3 and O_4 are themselves not marked mutex. In fact, the action mutex rules remain unchanged: two actions are marked mutex if either (a) one of the preconditions of one of the actions is mutex with one of the preconditions of the other action, or (b) if the unconditional effects of one of the actions deletes a precondition or an unconditional effect of the other action.

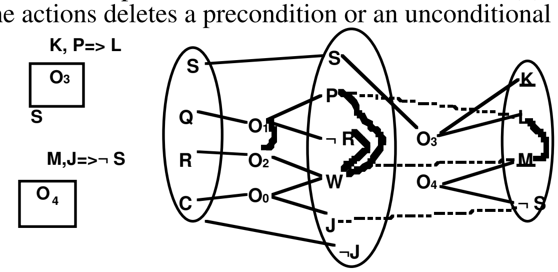

Figure 4. Example illustrating direct handling of conditional effects (mutexes shown as black links)

Coming back to our example, the goals K and M are present in the third level proposition list and backward search can commence. Since some of the goals may be supported by conditional establishments, in addition to the primary preconditions of the selected actions the backward search must also subgoal on the secondary causation preconditions, and further consider preservation preconditions for all the subgoals with respect to all the selected actions. In our example, K can be supported by O_3, and M can be supported by the conditional establishment [O_4, W]. Thus, we must subgoal on S, which is the precondition of O_3, as well as W, which is the causation precondition of M with respect to O_4. Furthermore, we must ensure that neither of the selected actions violate any of these subgoals. Since O_4 has a conditional effect ¬S, we must post the preservation precondition of S with respect to $O_{4,}$, which is ¬J. Since none of the selected actions violates ¬J, we can stop with the three subgoals S, W and ¬J. Since all of these can be supported at the next level through unconditional establishments with the resulting subgoals being true in the initial state, the backward search succeeds.

4.4 Improving Graphplan backward search using CSP techniques

The connection between the planning-graph and the dynamic constraint satisfaction problem suggests several possible improvements for the backward search phase of Graphplan. Considering such improvements is further motivated by the fact that compiling the planning-graph into a SAT instance and solving it using SAT solving techniques has been shown to out-perform standard Graphplan [Kautz & Selman, 1996; Bayardo & Schrag, 1997]. The current understanding in the CSP literature (c.f. [Frost & Dechter, 1996; Bayardo & Schrag, 1997]) is that the best systematic search algorithms for the standard CSP involve forward checking (a form of constraint propagation), dynamic variable ordering, dependency directed backtracking and no-good learning. Of these, the Graphplan backward search uses only memoization -- a limited form of no-good learning. Thus, it is worth considering the utility of the other three ideas as well as the full-fledged no-good learning. Supporting forward checking involves filtering out the conflicting actions from the domains of the remaining goals, as soon as a particular goal is assigned. Dynamic variable ordering involves selecting for assignment the goal that has the least number of remaining establishers. Preliminary results[7] show that these techniques can bring about significant improvements in Graphplan's performance. For example, Graphplan is able to solve a blocks world benchmark problem 8 times faster with the "least number of remaining establishers" variable ordering heuristic as compared with that of the "most number of remaining establishers." These results also show that contrary to Blum & Furst's speculations [1995] "goal ordering" does have a practical impact on Graphplan.

As discussed in [Kambhampati, 1997b], in general a failure explanation (no-good) for the dynamic CSP specifies a set of assigned variables (with their assignments) and a set of unassigned variables. The semantics of such a failure explanation are that if it is part of any search node produced in solving that DCSP, then none of the branches under that search node can lead to a solution. In this context, Graphplan memos can be seen as a subset of possible failure explanations -- those that only name unassigned variables. Failure explanations consisting of both assigned and unassigned variables can help in situations where most ways of achieving a set of subgoals at a level k fail, but a small fraction succeed. In such cases, no memos can be learned, and without generalized failure explanations Graphplan will be forced to repeat failing assignments many times. It would be interesting to investigate the utility of a full-fledged learning strategy, involving the explanation of failures at leaf nodes, and regression and propagation of leaf node failure explanations to compute interior node failure explanations, along the lines described in [Kambhampati, 1997b]. Not only does such a strategy promise to increase learning opportunities, it could also make the existing memos more "general". Specifically, Graphplan stores all of the goals of a failing goal set at a level k, even if the failure is actually due to the presence of only a small subset of them.[8] Full-fledged learning will help alleviate this situation, making the stored goal sets more likely to be useful in other branches of the search..

4.5 Teasing planning and scheduling apart in Graphplan

By now it is well-known that despite its efficiency, Graphplan can perform poorly on some problems. This can happen for one of two reasons -- first, due to irrelevant actions in the domain the planning-graph grows too fast, using up all existing memory. This can be handled to a large extent by making Graphplan goal-directed (see Sections 5.1 and 5.2). Another class of problems, exemplified by the benchmarks used in Kautz and Selman's SATPLAN experiments [1996] are hard because they require too much effort in the backward search phase. An analysis of these problems reveals that the main reason for the difficulty here is that Graphplan combines the planning and scheduling (resource allocation) phases, making the combined problem much harder to solve. To understand this, note that in a blocks world problem like `bw-large-a` in the SATPLAN suite, Graphplan finds a plan that is correct save the ordering of the actions in one of the early iterations, but spends enormous time in the next several iterations effectively "linearizing" this plan to handle the resource restrictions imposed by the availability of a single hand. The inefficiency of this approach is apparent when we realize that during these later iterations, Graphplan is not only looking at the approximately correct plan, but also the rest of the search space. This approach results in a degradation of Graphplan's performance when given additional resources (e.g., the number of robot hands are increased beyond 2 in blocks world) -- a decidedly counter-intuitive behavior. A much better alternative would thus involve separating the planning and scheduling phases in Graphplan, terminating planning as soon as a plan that is correct modulo resource allocation is found, and then starting a separate scheduling phase with this single plan as the

[7] This is part of joint work with Dan Weld.

[8] Notice that this is different from the "subset memoization" that is employed in the Graphplan implementation, which involves checking if any subset of the current goal set is matching a stored memo. Here we are interested in storing a smaller memo to begin with (so both normal and subset memoizations have a better chance of succeeding).

input. Our preliminary investigations show that such a strategy often leads to dramatic improvements in the efficiency of plan generation. We are currently in the process of completing the formalization and evaluation of this idea [Srivastava & Kambhampati, 1997].

5 Conclusion

In this paper, we have provided a rational reconstruction of Blum and Furst's Graphplan algorithm starting from forward state-space search. This reconstruction has shown that Graphplan's planning-graph data structure is an upper-bound approximation to the unioned representation of the state-space search tree, with the mutex propagation making the approximation finer. The backward search phase corresponds to solving a dynamic constraint satisfaction problem. We used the reconstruction to clarify the role and relative tradeoffs offered by "action parallelism" and "mutex propagation" in the Graphplan algorithm, and to interpret Graphplan as a refinement planner. We have also used our understanding to sketch a variety of ways of extending the Graphplan algorithm, including making Graphplan goal-directed, empowering it to handle more expressive action representations, supporting backward propagation of mutex information, improving backward search by exploiting CSP techniques and teasing apart planning and scheduling in Graphplan. Further details and evaluation of these extensions will be described in a forthcoming extended version of this paper.

References

Barrett, A. and Weld, D. 1994. Partial Order Planing: Evaluating possible efficiency gains. *Artificial Intelligence* , 67(1):71-112.

Blum, A. and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. IJCAI-95* (Extended version appears in *Artificial Intelligence, 90(1-2)*)

Bayardo, R. and Schrag, R. 1997. Using CSP look-back techniques to improve real-word SAT instances. In *Proc. AAAI-97.*

Frost, D and Dechter, R. 1994. In search of best constraint satisfaction search. In *Proc. AAAI-94*.

Drummond, M. 1989. Situated control rules. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning,* 103-113. Morgan Kaufmann.

Kambhampati, S., Knoblock, C., and Yang, Q. 1995. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence,* 76(1-2):167-238.

Kambhampati, S. and Yang, X. 1996, On the role of disjunctive representations and constraint propagation in refinement planning. In *Proceedings of the fifth International Conference on Principles of Knowledge Representation and Reasoning.* 35-147.

Kambhampati, S. 1997a. Challenges in bridging plan synthesis paradigms. In Proc. IJCAI-97.

Kambhampati, S. 1997b. On the relations between intelligent backtracking and explanation-based learning in planning and constraint satisfaction. *rakaposhi.eas.asu.edu/pub/rao/jour-ddb.ps.*

Kambhampati. S. 1997c. Refinement planning as a unifying framework for plan synthesis. *AI Magazine.* 8(2).

Kautz, H. and Selman, B. 1996. Pushing the envelope: Planning Propositional Logic and Stochastic Search. In *Proceedings of National Conference on Artificial Intelligence.* 1194-11201

McDermott, D. 1996. A Heuristic estimator for means-ends analysis in planning. In: *Proceedings of 3rd International Conference on AI Planning Systems. AAAI Press.* 142-149.

Mittal, S. and Falkenhainer, B. 1990. Dynamic Constraint Satisfaction Problems. In *Proc. AAAI-90.*

Parker, E and Kambhampati, S. 1997. Making Graphplan goal-directed. Forthcoming.

Reisig, W. 1982. *Petrie Nets: An Introduction.* Springer-Verlag.

Smith, D and Peot, M. 1996. Suspending recursion in partial order planning. In *Proc. 3rd Intl. Conference on AI Planning Systems.*

Srivastava, B and Kambhampati, S. 1997. Teasing apart planning and scheduling in Graphplan. Forthcoming.

Tsang, E. *Constraint Satisfaction*. Academic Press. 1993.