Improving Graphplan's search with EBL & DDB Techniques

Subbarao Kambhampati*
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
email: rao@asu. edu URL: rakaposhi.eas.asu.edu/yochan.html

Abstract

I highlight some inefficiencies of Graphplan's backward search algorithm, and describe how these can be eliminated by adding explanation-based learning and dependency-directed backtracking capabilities to Graphplan. I will then demonstrate the effectiveness of these augmentations by describing results of empirical studies that show dramatic improvements in run-time (w lOOx speedups) as well as solvability-horizons on benchmark problems across seven different domains.

1 Introduction

Graphplan [Blum & Furst, 97] is currently one of the more efficient algorithms for solving classical planning problems. Four of the five competing systems in the recent AIPS-98 planning competition were based on the Graphplan algorithm [McDermott, 98], Extending the efficiency of the Graphplan algorithm thus seems to be a worth-while activity. In this paper, I describe my experience with adding explanation-based learning (EBL) and dependency-directed backtracking capabilities (DDB) to Graphplan's backward search. Both EBL and DDB are based on explaining failures at the leaf-nodes of a search tree, and propagating those explanations upwards through the search tree [Kambhampati, 98]. DDB involves using the propagation of failure explanations to support intelligent backtracking, while EBL involves storing interior-node failure explanations, for pruning future search nodes. Graphplan does use a rudimentary form of failure-driven learning that it calls "memoization." As we shall see in this paper, Graphplan's brand of learning is quite weak as there is no explicit analysis of the reasons for failure. Instead the explanation of failure of a search node is taken to be *all* the constraints in that search node. As explained in [Kambhampati, 98], this not only eliminates the opportunities for dependency directed backtracking, it also adversely affects the utility of the stored memos.

*This research is supported in part by NSF young investigator award (NYI) IRI-9457634, ARPA/Rome Laboratory planning initiative grant F306G2-95-C-0247, Army AASERT grant DAAH04-96-1-0247, AFOSR grant F20602-98-1-0182 and NSF grant IRI-9801676.1 thank Maria Fox, Derek Long, Terry Zimmerman, Amol Mali and Biplav Srivastava for comments on this work.

Adding full-fledged EBL and DDB capabilities in effect gives Graphplan both the ability to do intelligent backtracking, and the ability to learn generalized memos that are more likely to be applicable in other situations. Technically, this involves generalizing conflict-directed backjumping [Prosser, 93], a well-known technique in CSP, to work in the context of Graphplan's backward search. Empirically, the EBL/DDB capabilities improve Graphplan's search efficiency quite dramatically-allowing it to easily solve several problems that have hither-to been hard or unsolvable. In particular, I will report on my experiments with the benchmark problems used in [Kautz & Selman, 96], as well as four other standard test domains. The experiments show up-to 100 fold speedups on individual problems.

This paper is organized as follows. In the next section, I provide some background on Graphplan's backward search. In Section 3,1 discuss some inefficiencies of the backtracking and learning methods used in normal Graphplan that motivate the need for EBL/DDB capabilities. Section 4 describes how EBL and DDB are added to Graphplan. Section 5 presents empirical studies demonstrating the usefulness of these augmentations. Section 6 discusses related work and Section 7 presents conclusions and some directions for further work.

2 Review of Graphplan algorithm

Graphplan algorithm [Blum & Furst, 97] can be seen as a "disjunctive" version of the forward state space planners [Kambhampati et. al., 97a]. It consists of two interleaved phases - a forward phase, where a data structure called "planning-graph" is incrementally extended, and a backward phase where the planning-graph is searched to extract a valid plan. The planning-graph (see Figure 1) consists of two alternating structures, called "proposition lists" and "action lists." Figure 1 shows a partial planning-graph structure. We start with the initial state as the zeroth level proposition list. Given a k level planning graph, the extension of structure to level k + 1 involves introducing all actions whose preconditions are present in the k^{th} level proposition list. In addition to the actions given in the domain model, we consider a set of dummy "persist" actions, one for each condition in the k^{th} level proposition list. A "persist-C" action has C as its precondition and C as its effect. Once the actions are introduced, the proposition list at level k + 1 is constructed as just the union of the effects of all the introduced actions. Planning-graph maintains the dependency links between the actions at

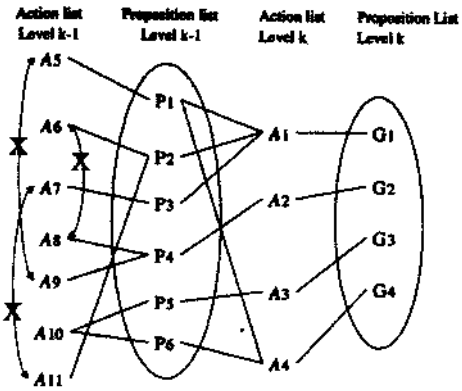

Figure 1: The running example used to illustrate EBL/DDB in Graphplan

level *k* +1 and their preconditions in level *k* proposition list and their effects in level *k* +1 proposition list. The planning-graph construction also involves computation and propagation of "mutex" constraints. The propagation starts at level 1, with the actions that are statically interfering with each other (i.e., their preconditions and effects are inconsistent) labeled mutex. Mutexes are then propagated from this level forward by using two simple propagation rules. In Figure 1, the curved lines with x-marks denote the mutex relations.

The search phase on a A: level planning-graph involves checking to see if there is a sub-graph of the planning-graph that corresponds to a valid solution to the problem. This involves starting with the propositions corresponding to goals at level *k* (if all the goals are not present, or if they are present but a pair of them are marked mutually exclusive, the search is abandoned right away, and planning-graph is grown another level). For each of the goal propositions, we then select an action from the level *k* action list that supports it, such that no two actions selected for supporting two different goals are mutually exclusive (if they are, we backtrack and try to change the selection of actions). At this point, we recursively call the same search process on the k—1 level planning-graph, with the preconditions of the actions selected at level *k* as the goals for the *k* — 1 level search. The search succeeds when we reach level 0 (corresponding to the initial state).

Previous work [Kambhampati et. al., 97a] had explicated the connections between this backward search phase of Graphplan algorithm and the constraint satisfaction problems (specifically, the dynamic constraint satisfaction problems, as introduced in [Mittal & Falkenhainer, 90]).

3 Some inefficiencies in Graphplan's backward search

To motivate the need for EBL and DDB, we shall first review the details of Graphplan's backward search, and pinpoint some of its inefficiencies. We shall base our discussion on the example planning graph from Figure 1. Assuming that G_1 - • • G_4 are the top level goals of the problem we are interested in solving, we start at level k, and select actions to support the goals G_1 • • • G_4. To keep matters simple, we shall assume that the search assigns the conditions (variables) at each level from top to bottom (i.e., G_1 first, then G_2 and so on). Further, when there is a choice in the actions (values) that can support a condition, we will consider the top actions first Since there is only one choice for each of the conditions at level k, and none of the actions are mutually exclusive with each other, we select the actions A_1, A_2, A_3 and A_4 for supporting the conditions at level *k*. We now have to make sure that the preconditions of A_1, A_2, A_3, A_4 are satisfied at level k - 1. We thus subgoal on the conditions P_1 • • P_6 at level *k* - 1, and recursively start the action selection for them. We select the action A_5 for P_1. For P_2, we have two supporting actions, and using our convention, we select A_6 first/For P_3, A_7 is the only choice. When we get down to selecting a support for P_4, we again have a choice. Suppose we select A_8 first. We find that this choice is infeasible as *As* is mutually exclusive with A_6 that is already chosen. So, we backtrack and choose A_9, and find that it too is mutually exclusive with a previously selected action, A_5. We now are stymied as there are no other choices for P_4. So, we have to backtrack and undo choices for the previous conditions. Graphplan uses a chronological backtracking approach, whereby, it first tries to see if P_3 can be re-assigned, and then P_2 and so on. Notice the first indication of inefficiency here - the failure to assign P_4 had nothing to do with the assignment for P_3, and yet, chronological backtracking will try to re-assign P_3 in the vain hope of averting the failure. This can lead to a large amount of wasted effort had it been the case that P_3 did indeed have other choices.

As it turns out, we find that P_3 has no other choices and backtrack over it. P_2 does have another choice - A_{11}. We try to continue the search forward with this value for P_2, but hit an impasse at P_3-since the only value of P_3, A_7 is mutex with A_{11} • At this point, we backtrack over P_3, and continue backtracking over P_2 and P_1, as they too have no other remaining choices. When we backtrack over P_1, we need to go back to level *k* and try to re-assign the goals at that level. Before this is done, the Graphplan search algorithm makes a "memo" signifying the fact that it failed to satisfy the goals P_1 • • P_6 at this level, with the hope that if the search ever subgoals on these same set of goals in future, we can scuttle it right away with the help of the remembered memo. Here is the second indication of inefficiency- we are remembering all the subgoals P_1 • • • P_6 even though we can see that the problem lies in trying to assign P_1, P_2, P_3 and P_4 simultaneously, and has nothing to do with the other subgoals. If we remember {P_1, P_2, P_3, P_4} as the memo as against {P_1 • • • P_6}, the remembered memo would be more general, and would have a much better chance of being useful in the future.

After the memo is stored, the backtracking continues into level *k* - once again in a chronological fashion, trying to reassign G_4,G_3,(G_2 and G_1 in that order. Here we see the third indication of inefficiency caused by chronological backtracking - G_3 really has no role in the failure we encountered in assigning P_3 and P_4 - since it only spawns the condition P_5 at level *k* - 1. Yet, the backtracking scheme of Graphplan considers reassigning G_3. A somewhat more subtle point is that reassigning G_4 is not going to avert the failure either. Although G_4 requires P_1 one of the conditions taking part in the failure, P_1 is also required by G_1 and unless G_1 gets reassigned, considering further assignments to G_4 is not going to avert the failure.

For this example, we continue backtracking over G_2 and

G_1 too, since they too have no alternative supports, and finally memoize $\{G_1, G_2, G_3, G_4\}$ at this level. At this point the backward search fails, and Graphplan extends the planning graph by another level before re-initiating the backward search on the extended graph.

4 Improving backward search with EBL and DDB

I will now describe how Graphplan's backward search can be augmented with full fledged EBL and DDB capabilities to eliminate the inefficiencies pointed out in the previous section. Informally, EBL/DDB strategies involve 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, 98]. The specific extensions I propose to the backward search can essentially be seen as adapting conflict-directed backjumping strategy [Prosser, 93], and generalizing it to work with Graphplan's backward search (which can be seen as a form of dynamic constraint satisfaction problem). The development here parallels the framework described in [Kambhampati, 98].

The algorithm is shown in pseudo-code form in Figure 2. It contains two mutually recursive procedures `find-plan` and `assign-goals`. The former is called once for each level of the planning-graph. It then calls `assign-goals` to assign values to all the required conditions at that level, `assign-goals` picks a condition, selects a value for it, and recursively calls itself with the remaining conditions. When it is invoked with an empty set of conditions to be assigned, it calls `find-plan` to initiate the search at the next (previous) level.

In order to illustrate how EBL/DDB capabilities are added, let's retrace the previous example, and pick up at the point where we are about to assign P_4 at level k — 1, having assigned P_1, P_2 and P_3. When we try to assign the value A_5 to P_4, we violate the mutex constraint between A_6 and A_8. An explanation of failure for a search node is a set of constraints from which *False* can be derived. The complete explanation for this failure can thus be stated as:

$$P_2 = A_6 \wedge P_4 = A_8 \wedge (P_2 = A_6 \Rightarrow P_4 \neq A_8)$$

Of this, the part $P_2 = A_6 \Rightarrow P_4 \neq A_8$ can be stripped from the explanation since the mutual exclusion relation will hold as long as we are solving this particular problem with these particular actions. Further, we can take a cue from the conflict directed back-jumping algorithm [Prosser, 93], and represent the remaining explanation compactly in terms of "conflict sets." Specifically, whenever the search reaches a condition *c* (and is about to find an assignment for it), its conflict set is initialized as {c}. Whenever one of the possible assignments to *c* is inconsistent (mutually exclusive) with the current assignment of a previous variable c', we add c' to the conflict set of c. In the current example, we start with $\{P_4\}$ as the conflict set of P_4, and expand it by adding P_2 after we find that A_5 cannot be assigned to P_4 because of the choice of A_6 to support P_2. Informally, the conflict set representation can be seen as an incrementally maintained (partial) explanation of failure, indicating that there is a conflict between the current value of P_2 and one of the possible values of P_4 [Kambhampati, 98].

```
Find- Plan( G:goals,  pg: plan graph, k: level)

 If k=0, Return an empty     subplan P with success.
 If there is a memo M that is a subset of      G,
    Fail, and return M as the conflict set
 Call Assign- goals(G, pg, k, nil).
  If Assign-goals fails and     returns a conflict set M,
     Store M as a memo
     Regress M over actions selected at level k+1 to get R
     Fail and return R as the conflict set
  If Assign-goals succeeds, and returns a k-level      subplan P,
     Return P with success

Assign- goals( G:goals,  pg: plan graph, k: level, A: actions)

 If G is empty,
   Let U union of preconditions of the actions in A
   Call Find- plan( U, pg, k-1 )
     If it fails and returns a conflict set      R,
        Fail and return  R
      If it succeeds and returns a     subplan  P of length k-1
        Succeed and return a k length      subplan  P.A

 If G is not empty,
   Select a goal   g from  G
   Let cs = { g }, and Ag be the set of actions from level     k in
        pg  that support  g
   L1: If  Ag is empty, Fail and return     cs as the conflict set
        Else, pick an action     a from  Ag
          If a is mutually exclusive with some action      b in A
            Let l be the goal that   b was selected to support
           Set   cs = cs + { l }
            Goto L1
          Else (  a is not mutually exclusive with any action in      A)
            Call Assign-   goals( G -{ g },  pg, k , A +{a} )
              If the call fails and returns a conflict set      C
                 If g is part of  C
                  Set    cs = cs + C   ;conflict set absorption
                    Goto L1
                Else (   g is not part of  C)
                   Fail and return      C as the conflict set
                                ;dependency directed    backjumping
```

Figure 2: A pseudo-code description of Graphplan backward search enhanced with EBL/DDB capabilities.

We now consider the second possible value of P_4, viz., A_9, and find that it is mutually exclusive with A_5 which is currently supporting P_1. Following our practice, we add P_1 to the conflict set of P_4. At this point, there are no further choices for P_4, and so we backtrack from P_4, passing the conflict set of P_4, viz., $\{P_1, P_2, P_4\}$ as the reason for its failure. In essence, the conflict set is a shorthand notation for the following complete failure explanation [Kambhampati, 98] r[1]

$$[(P_4 = A_8) \vee (P_4 = A_9)] \wedge (P_1 = A_5 \Rightarrow P_4 \neq A_9) \wedge (P_2 = A_6 \Rightarrow P_4 \neq A_8) \wedge P_1 = A_5 \wedge P_2 = A_6$$

It is worth noting at this point that when P_4 is revisited in the future (with different assignments to the preceding variables), its conflict set will be re-initialized to $\{P_4\}$ before

[1]We strip the first (disjunctive) clause since it is present in the graph structure, and the next two implicative clauses since they are part of the mutual exclusion relations that will not change for this problem. The conflict set representation just keeps the condition (variable) names of the last two clauses - denoting, in essence, that it is the current assignments of the variables P_1 and P_2 that are causing the failure to assign P_4.

considering any assignments to it

Dependency directed backtracking: The first advantage of maintaining the conflict set is that it allows a transparent way of dependency directed backtracking [Kambhampati, 98]. In the current example, having failed to assign P_4, we have to start backtracking. We do not need to do this in a chronological fashion however. Instead, we jump back to the most recent variable (condition) taking part in the conflict set of P_4 - in this case P_2. By doing so, we are avoiding considering other alternatives at P_3, and thus avoiding one of the inefficiencies of the standard backward search. It is easy to see that such back-jumping is sound since P_3 is not causing the failure at P_4 and thus re-assigning it won't avert the failure.

Continuing along, whenever the search backtracks to a condition c, the backtrack conflict is absorbed into the current conflict set of c. In our example, we absorb $\{P_1, P_2, P_4\}$ into the conflict set of P_2, which is currently {P_2} (making $\{P_1, P_2, P_4\}$ the new conflict set of P_2)1. We now assign A_{11}, the only remaining value, to P_2, Next we try to assign P_3 and find that its only value A_7 is mutex with A_{11}. Thus, we set conflict set of P_3 to be $\{P_3, P_2\}$ and backtrack with this conflict set. When the backtracking reaches P_2, this conflict set is absorbed into the current conflict set of P_2 (as described earlier), giving rise to $\{P_1, P_2, P_3, P_4\}$ as the current combined failure reason for P_2. This step illustrates how the conflict set of a condition is incrementally expanded to collect the reasons for failure of the various possible values of the condition.

At this point, P_2 has no further choices, so we backtrack over P_2 with its current conflict set, $\{P_1, P_2, P_3, P_4\}$. At P_1, we first absorb the conflict set $\{P_1, P_2, P_3, P_4\}$ into P_1's current conflict set, and then re-initiate backtracking since P_1 has no further choices.

Now, we have reached the end of the current level *(k* - 1). Any backtracking over P_1 must involve undoing assignments of the conditions at the k^{th} level. Before we do that however, we carry out two steps: memoization and regression.

Memoization: Before we backtrack over the first assigned variable at a given level, we store the conflict set of that variable as a memo at that level. In the current example, we store the conflict set $\{P_1, P_2, P_3, P_4\}$ of P_1 as a memo at this level. Notice that the memo we store is shorter (and thus more general) than the one stored by the normal Graphplan, as we do not include P_5 and P_6, which did not have anything to do with the failure.[2]

Regression: Before we backtrack out of level *k* - 1 to level k, we need to convert the conflict set of (the first assigned variable in) level k-1 so that it refers to the conditions in level *k.* This conversion process involves regressing the conflict set over the actions selected at the k^{th} level [Kambhampati, 98]. In essence, the regression step computes the (smallest) set of conditions (variables) at the k^{th} level whose supporting actions spawned (activated, in DCSP terms) the conditions (variables) in the conflict set at level *k* - 1. In the current case, our conflict set is $\{P_1, P_2, P_3, P_4\}$. We can see that P_2, P_3 are required because of the condition G_1 at level *k,* and the condition P_4 is required because of the condition G_2.

In the case of condition P_1, both G_1 and G_4 are responsible for it, as both their supporting actions need P_1. In such cases we have two heuristics for computing the regression: (1) Prefer choices that help the conflict set to regress to a smaller set of conditions (2) If we still have a choice between multiple conditions at level *k,* pick the one that has been assigned[1] earlier. The motivation for the first rule is to keep the failure explanations as compact (and thus as general) as possible, and the motivation for the second rule is to support deeper dependency directed backtracking. It is important to note that these heuristics are aimed at improving the performance of the EBL/DDB and do not affect the soundness and completeness of the approach.

In the current example, the first of these heuristics applies, since P_1 is already required by G_1, which is also requiring P_2 and P_3. Even if this was not the case (i.e., G_1 only required P_1), we still would have selected G_1 over G_4 as the regression of P_1, since G_1 was assigned earlier in the search.

The result of regressing $\{P_1, P_2, P_3, P_4\}$ over the actions at k^{th} level is thus $\{G_1, G_2\}$. We start backtracking at level *k* with this as the conflict set. We jump back to G_2 right away, since it is the most recent variable named in the conflict set. This avoids the inefficiency of re-considering the choices at G_3 and G_4, as done by the normal backward search. At G_2, the backtrack conflict set is absorbed, and the backtracking continues since there are no other choices. Same procedure is repeated at G_1. At this point, we are once again at the end of a level-and we memoize $\{G_1, G_2\}$ as the memo at level *k.* Since there are no other levels to backtrack to, Graphplan is called on to extend the planning-graph by one more level.

Notice that the memos based on EBL analysis capture failures that may require a significant amount of search to rediscover. In our example, we are able to discover that $\{G_1, G_2\}$ is a failing goal set despite the fact that there are no mutex relations between the choices of the goals G_1 and G_2

Using the Memos (EBL): Before we end this section, there are a couple of observations regarding the use of the stored memos. In the standard Graphplan, memos at each level are stored in a level-specific hash table. Whenever backward search reaches a level *k* with a set of conditions to be satisfied, it consults the hash table to see if this exact set of conditions is stored as a memo. Search is terminated only if an exact hit occurs. Since EBL analysis allows us to store compact memos, it is not likely that a complete goal set at some level *k* is going to exactly match a stored memo. What is more likely is that a stored memo is a subset of the goal set at level *k* (which is sufficient to declare that goal set a failure). In other words, the memo checking routine in Graphplan needs to be modified so that it checks to see if some subset of the current goal set is stored as a memo. The naive way of doing it - which involves enumerating all the subsets of the current goal set and checking if any of them are in the hash table- turns out to be very costly. One needs more efficient data structures, such as the set-enumeration trees [Rymon, 92]. Indeed, Koehler and her co-workers [Koehler et. al., 97] have developed a (seeming variation of set-enumeration tree) data structure called UB-Trees for storing the memos. The UB-Tree structures can efficiently check if any subset of the current goal set has been stored as a memo.

[2] While in the current example, the memo includes all the conditions up to P4 (which is the farthest we have gone in this level), even this is not always necessary. We can verify that P_3 would not have been in the memo set if A_{11} were not one of the supporters of P_2.

Problem	Graphplan with EBL/DDB					Normal Graphplan					Speedup
	Tt	Mt	# Btks	AvLn	AvFM	Tt.	Mt.	# Btks	AvLn	AvFM	
Huge-Fact (18/18)	8.20	0.73	2004K	9.52	2.52	12.9	0.55	5181K	11.3	1.26	1.6x
BW-Large-B (18/18)	4.77	0.33	798K	10.15	3.32	9.8	0.14	2823K	11.83	1.13	2x
Rocket-ext-a (7/36)	2.26	1.0	764K	8.5	82	53.21	31.24	8128K	23.9	3.2	23x
Rocket-ext-b (7/36)	2.5	1.4	569K	7.5	101	40	21	10434K	23.8	3.22	16x
Att-log-a (11/79)	5.9	2.67	2186K	8.21	46.18	>4hr	-	-	32	-	>40x
Gripper-6 (11/17)	0.3	0.1	201K	6.9	6.2	3.39	1.15	2802K	14.9	4.9	12x
Gripper-8 (15/23)	7.14	2.6	4426K	9	7.64	>4hr	-	-	17.8	-	>33x
Tower-5 (31/31)	0.4	0.05	277K	6.7	2.7	21.4	3.73	19070K	20.9	2.2	48x
Tower-6 (63/63)	8.3	0.6	4098K	7.9	2.8	>4hr	-	-	22.3	-	>29x
Ferry-41 (27/27)	1.29	0.38	723K	7.9	2.54	122.5	76.4	33341K	24.5	2.3	95x
Ferry-5 (31/31)	3.7	1.6	1993K	8.8	2.53	>4hr	-	-	197x	-	>65x
Tsp-10 (10/10)	2.9	0.67	2232K	6.9	12	217	123	69974K	13	5	75x

Table 1: Empirical performance of EBL/DDB. Unless otherwise noted, times are in cpu minutes on a spare ultra 1 with 128 meg RAM, running allegro common lisp compiled for speed. "Tt" is total time, "Mt" is the time used in checking memos and "Btks" is the number of backtracks done during search. The numbers in parentheses next to the problem names list the number of time steps and number of actions respectively in the solution. AvLn and AvFM denote the average memo length and average number of failures detected per stored memo respectively.

The second observation regarding memos is that they can often serve as a failure explanation in themselves. Suppose we are at some level k, and find that the goal set at this level subsumes some stored memo M. We can then use *M* as the failure explanation for this level, and regress it back to the previous level. Such a process can provide us with valuable opportunities for further back jumping at levels above *k.* It also allows us to learn new compact memos at those levels. Note that none of this would have been possible with normal memos stored by Graphplan, as the only way a memo can declare a goal set at level *k* as failing is if the memo is exactly equal to the goal set. In such a case regression will just get us all the goals at level *k* 4- 1, and does not buy us any back-jumping or learning power [Kambhampati, 98].

5 Empirical Evaluation

We have now seen the way EBL and DDB capabilities are added to the backward search by maintaining and updating conflict-sets. We also noted that EBL and DDB capabilities avoid a variety of inefficiencies in the standard Graphplan backward search. That these augmentations are soundness and completeness preserving follows from the corresponding properties of conflict-directed back-jumping [Kambhampati, 98]. The remaining (million-dollar) question is whether these capabilities make a difference in practice. I now present a set of empirical results to answer this question.

I implemented the EBL/DDB approach described in the previous section cm top of a Graphplan implementation in lisp.[3] The changes needed to the code to add EBL/DDB capability were minor - only two functions As `sign-goals` and `find-plannneeded` non-trivial changes. I also added the UB-Tree subset memo checking code described in [Koehler et. al., 97]. I then ran several comparative experiments on the "benchmark" problems from [Kautz & Selman, 96], as well as from four other domains. The specific domains included blocks world, rocket world, logistics domain, gripper domain, ferry domain, traveling salesperson domain, and towers of hanoi. The specifications of the problems as well as domains are publicly available. Table 1 shows the statistics on the times taken and number of backtracks made by normal Graphplan, and Graphplan with EBL/DDB capabilities.

Runtime Reductions & Solvability improvements: The first thing we note is that EBL/DDB techniques can offer quite dramatic speedups - upto 100x in the seven domains I tested. We also note that the number of backtracks reduces significantly and consistently with EBL/DDB. Moreover, EBL/DDB techniques push the solvability horizon as many problems were unsolvable without EBL even after 3-4 hours of cpu time![4]

Reduction in Memo Length: The results also show that as expected the length of memos stored by Graphplan decreased substantially when EBL/DDB strategies are employed. For example, the average memo length (the column named "AvLn" in Table 1) goes down from 32 to 8 in logistics, and 24 to 8 in the ferry domain. Furthermore, the relative reductions in memo length in different domains are well correlated with the speedups seen in those domains. Specifically, we note that blocks world domain, which shows a somewhat lower (~2x) speedup also has lower memo-length reduction (from 11.8 to 10). Similarly, the fact that the average length of memos for rocket-ext-a problem is 8.5 with EBL, and 24 without EBL, shows in essence that normal Graphplan is re-discovering an 8-sized failure embedded in many many possible ways in a 24 sized goal set - storing a new memo each time (incurring both increased backtracking and matching costs)! It is thus no wonder that normal Graphplan performs badly compared to Graphplan with EBL/DDB.

Utility of stored memos: Since EBL/DDB store more gen-

[3]The original lisp implementation of Graphplan was done by Marie Peot The implementation was subsequently improved by David Smith.

[4]For our lisp system configuration, this amounted to about 12-20 hours of real time including garbage collection time.

eral (smaller) memos than normal Graphplan, they should, in theory, generate fewer memos and use them more often. The columns labeled "AvFM" give the ratio of the number of failures discovered through the use of memos to the number of memos generated in the first place. This can be seen as a *rough* measure of the average "utility" of the stored memos. We note that the utility is consistently higher with EBL/DDB in all the solved problems. As an example, in rocket-ext-b, we see that on the average an EBL/DDB generated memo was used to discover failures 101 times, while the number was only 3.2 for the memos generated by the normal Graphplan.

The C vs. Lisp question: Given that most existing implementations of Graphplan are done in C with many optimizations, one nagging doubt is whether the dramatic speedups due to EBL/DDB are somehow dependent on the moderately optimized lisp implementation I have used in my experiments. Thankfully, the EBL/DDB techniques described in this paper have also been (re)implemented by Maria Fox and Derek Long on their STAN system. STAN is a highly optimized implementation of Graphplan that fared well in the recent AIFS planning competition. They have found that EBL/DDB resulted in the same dramatic speedups on their system too [Fox, 98].

6 Related Work

In their original implementation of Graphplan, Blum and Furst experimented with a variation of the memoization strategy called "subset memoization". In this strategy, they keep the memo generation techniques the same, but change the way memos are used, declaring a failure when a stored memo is found to be a subset of the current goal set. Since complete subset checking is costly, they experimented with a "partial" subset memoization where only subsets of length n and *n* - 1 are considered for an n sized goal set.

As we mentioned earlier, Koehler and her co-workers [Koehler et. al., 97] have re-visited the subset memoization strategy, and developed a more effective solution to complete subset checking that involves storing the memos in a data structure called UB-Tree, instead of in hash tables. The results from their own experiments with subset memoization, as well as my replication of those experiments [Kambhampati, 98b], are mixed at best, and indicate that the improvements are nowhere near those achievable through DDB and EBL. The reason for this is quite easy to understand - while they improved the memo checking time with the UB-Tree data structure, they are still generating and storing the same old long memos. In contrast, the EBL/DDB extension described here supports dependency directed backtracking, and by reducing the average length of stored memos, increases their utility significantly, thus offering dramatic speedups.

[Kambhampati, 98] describes the general principles underlying the EBL/DDB techniques and sketches how they can be extended to dynamic constraint satisfaction problems. [Kambhampati, 97a] discusses the relations between Graphplan and dynamic CSP problems. The development in this paper can be seen as an application of the ideas from these two papers.

7 Conclusion and Future work

In this paper, I motivated the need for adding EBL/DDB capabilities to Graphplan's backward search, and described the changes needed to support these capabilities in Graphplan. I also presented and analyzed empirical results to demonstrate that EBL/DDB capabilities significantly enhance the efficiency of Graphplan on benchmark problems in seven different domains. Based on these results, and the fact that I have not encountered any problems where EBL/DDB turned out to be a significant net drain on efficiency, I conclude that EBL/DDB extensions are very useful for Graphplan.

There are several ways in which this work can be (and is being) extended. Recently, I have added forward checking and dynamic variable ordering capabilities on top of EBL & DDB in Graphplan, and initial results show further improvements (up to a factor of 4) in run time [Kambhampati, 98b]. The success of EBL/DDB approaches in Graphplan is in part due to the high degree of redundancy in the planning graph structure. In [Zimmerman & Kambhampati, 99], we investigate techniques that are even more aggressive in exploiting this redundancy to improve planning performance. Finally, we are also studying the utility of memo-forgetting strategies such as relevance based learning [Bayardo & Schrag, 97], as well as inter-problem memo transfer.

References

Blum, A. and Furst, M. 1997. Fast planning through planning graph analysis. Artificial Intelligence, 90(1-2).

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

Fox, M, 1998 Private correspondence. Fall 1998.

Kambhampati, S., Parker, E., and Lambrecht, E., 1997a Understanding and Extending Graphplan. In Proceedings of 4th European Conference on Planning.

Kambhampati, S., Katukam, S. and Qu, Y, 1997b Failure driven Dynamic Search Control for Partial Order Planners: An Explanation-based approach", Artificial Intelligence. 88(l-2):253-315. 1997.

Kambhampati, S. 1998. On the relations between intelligent backtracking and explanation-based learning in planning and constraint satisfaction. *Artificial Intelligence.* Vol. 105, No. 1-2.

Kambhampati, S., 1998b EBL and DDB for Graphplan. ASU-CSE-TR 98-008. *rakaposhLeas.asu.edu/gp-ebl'tr.ps*

Koehler, J., Nebel, B., Hoffmann, J., and Dimopoulos, Y. 1997 Extending planning graphs to an ADL Subset. Technical Report No. 88. Albert Ludwigs University.

Kautz, H. & Selman, B. 1996. Pushing the envelope: Planning Proposirional Logic and Stochastic Search. In Proceedings of National Conference on Artificial Intelligence.

McDermott, D. 1998 AIPS-98 Planning Competition Results. URL: ftp.cs.yale.edu puh/mcdennc^dpscomp-results.html.

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

Prosser, P. 1993 . Domain filtering can degrade intelligent backtracking search. In Proc. UCAI, 1993.

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

Rymon, 1992 Set Enumeration Trees. In Proc. KR-92.

Zimmerman, T. & Kambhampati, S., 1999 Exploiting symmetry in the planning-graph via explanation-guided search. In Proc. AAAI-99.