Exploiting Symmetry in the Planning graph via Explanation-Guided Search

Terry Zimmerman & Subbarao Kambhampati*
Department of Computer Science & Engineering
Arizona State University, Tempe AZ 85287
Email: {zim,rao}@asu.edu URL: rakaposhi.eas.asu.edu/yochan.html

Abstract

We present a method for exploiting the symmetry in the planning graph structure and certain redundancies inherent in the Graphplan algorithm, so as to improve its backward search. The main insight underlying our method is that due to these features the backward search conducted at level k + 1 of the graph is essentially a replay of the search conducted at the previous level k with certain well-defined extensions. Our method consists of maintaining a pilot explanation structure capturing the failures encountered at previous levels of the search, and using it in an intelligent way to guide the search at the newer levels. The standard EBL and DDB techniques can be employed to control the size of the pilot explanation. The technique has been implemented in the EGBG system, and we present a preliminary empirical study.

1 Introduction

The advent of Graphplan [Blum & Furst, 95] as one of the fastest programs for solving classical planning problems, marked a significant departure from the planning algorithms studied up to that time. Recently efforts have been made to place the approach in perspective and investigate the applicability of a variety of speed-up techniques that have been proven effective in other planning and search-based problem-solving systems.

Graphplan conducts it's problem solution search by interleaving two distinct phases: a forward phase the builds a "planning graph" structure followed by a phase that conducts backward search on that structure. As it turns out, the planning graph contains a high degree of redundancy and symmetry suggesting several avenues for speeding up the search by exploiting these features.

This paper describes initial work with a novel approach that takes full advantage of the particular symmetry of the planning graph and Graphplan's backward search. During each backward search episode a concise trace structure (termed the 'pilot explanation') is generated to capture the key features of the search and this is then carried forward (assuming the search fails to find a solution) to direct subsequent search at all future levels of the planning graph. The pilot explanation thus acts a sort of memory of the previous search experience and, due to the inherent symmetry of the planning graph, closely models the search that Graphplan in fact undertakes at the next level. This trace structure prescribes the minimal set of constraints (search backtrack points) that need to be checked based on the experience at the previous level. Should all these constraints also hold at the current level the planning graph can immediately be extended to a new level. At every point where a constraint no longer holds backward search is resumed and the pilot explanation extended to reflect the experience gained

This approach, then, seeks to speedup Graphplan by avoiding all redundant search effort at each level of the planning graph at the cost of building, extending, and using the trace structure that is the pilot explanation. The idea of building an entire trace of the experience during a search process so as to later shortcut the effort needed to solve a similar problem is, of course, an old one (c.f. [Veloso, 94; Ihrig & Kambhampati, 97]). For most problems of real interest the sheer size of the trace structure makes the approach ineffective. It's the symmetry of the planning graph and the search that Graphplan conducts at each level that makes such a technique even feasible for this system. And even then, great care must be take to retain only essential search information in the trace structure and to control its growth if the "larger" planning problems of interest are to be addressed.

The rest of this paper is organized as follows. Section 2 provides a brief review of the Graphplan algorithm. Section 3 presents our approach for guiding backward search with the pilot explanation. This section first considers the sources of redundancy in the Graphplan, and explains how the representation and use of the pilot explanation is geared towards exploiting this redundancy. Section 4 describes results of preliminary experiments with EGBG, a version of Graphplan that uses the pilot explanation structure. Section 5 discusses several ways of improving the representation of the pilot explanation, by complementing the Graphplan's search algorithm with Explanation-based learning and Dependency directed backtracking. Section 6 discusses related work and Section 7 presents our conclusions.

2 Overview of Graphplan

Graphplan algorithm [Blum & Furst, 97] can be seen as a ``disjunctive'' version of the forward state space planners [Kambhampati et. al., 97]. 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

* This research is supported in part by NSF young investigator award (NYI) IRI-9457634, ARPA/Rome Laboratory initiative grant F30602-95-C-0247, Army AASERT grant DAAH04-96-1-0247, AFOSR grant F20602-98-1-0182 and NSF grant IRI-9801676.

graph is searched to extract a valid plan. The planning graph (see Fig. 1) consists of two alternating structures, called proposition lists and action lists. Fig. 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. The planning graph maintains the dependency links between the actions at 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 through the use of two simple propagation rules. In Figure 1, the curved lines with x-marks denote the mutex relations: two propositions at level k are marked mutex if all actions at level k that support one proposition are mutex with all actions that support the second proposition. Two actions at level k+1 are mutex if they are statically interfering or if one of the propositions (preconditions) supporting the first action is mutually exclusive with one of the propositions supporting the second action. It should be noted that mutex checking forms an integral part of both the graph building phase and the backward search phase, and is a major contributor to the total cpu time Graphplan may spend on a problem.

The search phase on a k 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 is 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). Once all goals for a level are supported, 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).

A final aspect of Graphplan's search is that when a set of (sub)goals for a level k is determined to be unsolvable, they are *memoized* at that level in a hash table. Correspondingly, when the backward search process later enters level k with a set of subgoals, they are first checked against the hash table to see if they've already been proved unsolvable.

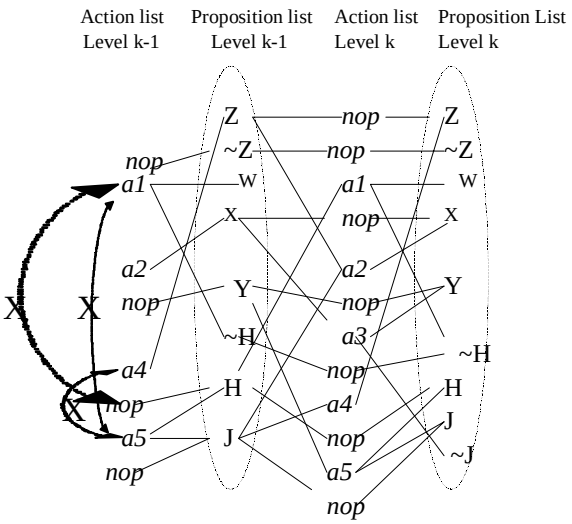

Figure 1. Example planning graph showing action and proposition levels and some of the mutex action pairs

3 Explanation-Guided Backward Search

We first describe the structural symmetry inherent in the planning graph and the use of the pilot explanation to take advantage of it during the backward search phase. The overhead entailed by such a system and the resulting tradeoffs that must be considered are then discussed.

3.1 Taking advantage of planning graph symmetry

Due to the inherent symmetry of the planning graph the backward search conducted at level k + 1 of the graph is essentially a replay of the search conducted at the previous level k *with certain well-defined extensions* [Ihrig & Kambhampati, 97]. When backward search at level k fails to find a solution the technique proposed here captures all the key features of the search episode in a special trace structure (termed the pilot explanation) which is then carried forward to direct the search at all future levels of the planning graph. At successively higher levels the trace structure prescribes the minimal set of constraints (search backtrack points) that need to be checked based on the experience at the previous level. Should all these constraints also hold at the current level the planning graph can immediately be extended to a new level. At every point where a constraint no longer holds backward search is resumed and the pilot explanation extended to reflect the experience gained

The following symmetrical or redundant features of the planning graph suggest possible shortcuts in the search process conducted at each new level:

- The proposition goal set that is to be satisfied at a new level k is exactly the same set that will be searched on at level k+1 when the planning graph is extended. That is, once the goal proposition set is present at level k it will be present at all future levels.

- The set of actions that can establish a given proposition at level k+1 always include the set establishing the proposition at level k and *may* include some newly added actions.
- The "constraints" (mutexes) that are active do not vary randomly over successive levels. A mutex that is active at level k may or may not continue to be active at level k+1, but once it becomes inactive at a level it never gets re-activated at future levels. For example, when a new action A1 is introduced at level k it's mutex status with every other action (in pair-wise fashion) at that level is determined. If it is mutex with A4 the pair may eventually become non-mutex at a future level, but thereafter they will remain non-mutex. And if A1 is initially non-mutex with A3 at level k it will never become mutex at higher levels.
- Two actions in a level that are "statically" mutex (i.e. their effects or preconditions conflict with each other) will be mutex at *all* succeeding levels

These factors, taken together, are responsible for considerable similarity (i.e. redundancy) in Graphplan's search performed at each successive planning graph level. Figure 2 illustrates the trace of such a search episode for the problem in Figure 1. The goals/subgoals to be satisfied as the backward search proceeds to each lower planning graph level are indicated within the circular nodes while the actions assigned to provide these goals are indicated by arrows. The points at which the search must backtrack (static or dynamic mutex action pairs or subgoals that are nogoods) are also shown.

The search trace in figure 2 can actually be viewed as an "explanation" for why search failed at level k or, alternately, why W X Y Z may be saved as a nogood at level k. Consider the situation once this search episode is complete and Graphplan is ready to add level k+1 to the graph and conduct backward search again at the new level. The search trace that will result from Graphplan's attempt to satisfy these goals beginning at level k+1 will closely resemble the search conducted at level k. In fact the *same search trace applies at level k+1 as long as the following 3 conditions hold*:

1. The action mutexes in the explanation for level k are *still* mutex for level k+1
2. There are no new actions establishing the goal propositions appearing in the explanation that were not also present (and explained away) at level k
3. Any nogoods appearing in the k-level failure explanation are also valid at the respective levels they map to in the search beginning at level k+1.

If these 3 conditions should happen to also hold for level k+1 we can immediately extend the graph to level k+2 because the goal set is also a nogood for level k+1. In this case the pilot explanation is carried forward to level k+2 unchanged.

A much more likely scenario will be that some number of the conditions will no longer hold when the trace is applied to level k+1. We can use the level k search trace to actually direct level k+1 search in a sound manner as long as each instance of one of these 3 conditions that appears in the trace is checked at the new level. If the constraint or condition holds no action is required. For each condition that does *not* hold the backward search must be re-

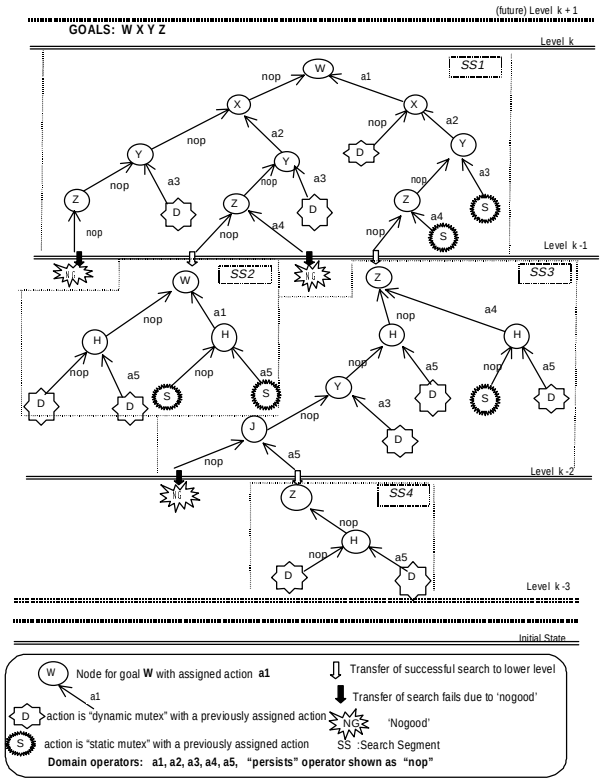

Figure 2. CSP-style trace of backward search at level k of the planning graph

sumed under the search parameters corresponding to the backtrack point in the search trace. Such partial search episodes will either find a solution or generate additional trace to augment the parent trace.

Because this specialized search trace can be used to direct *all* future backward search for this problem we refer to it as the ***pilot explanation (PE)***.

Note that in order to be sound the proposed system using the pilot explanation must revalidate *all* of the same constraints that standard Graphplan search would encounter at the points where it is forced to backtrack. So where is the hoped for search speedup? There are 3 primary sources:

1. *None* of the many successful action assignments made during the search that builds the PE have to be re-assigned during search at the next planning graph level. In general each of these assignments involved multiple mutex checks (with the previously assigned actions) and, as mentioned in Section 2, it is mutex checking that accounts for most of the cpu time Graphplan expends in a given search episode.
2. *None* of the many nogood checks made by Graphplan in the search embodied by the PE have to be repeated. This includes both the situations in which the subgoals were found in the memoized entries for the lower level (requiring backtracking) and situations in which the subgoals are not (initially) found to be nogood (so backward search continues at the lower level) Justification for this economy is given in the next section.

search segment (SS) :encapsulates the backward search conducted during *one* instance of search within a plan graph level
SS-goals: the goals to be established for this level and path
SS-assigns: results of attempts to satisfy the SS goals, grouped on a goal-by-goal basis. Possible results are: '**ok**' -for successful action assign, **num** -(an integer) action is dynamic mutex with the numth action assign attempted, or '**nt**' -no need to test: for static mutex, nogood, or transfer to next level
SS-transin: list of transfer-in events for transfers to this search segment from a higher level
pilot explanation (PE) : a multi-level list of search segments encapsulating only the dynamic features of previous backward search episodes that must be checked after the next plan graph extension. The single top level search segment at the front of the PE list is designated level 0, the last sublist in the PE list contains all search segments at the deepest level reached in previous search (level: (length PE) – 1). For example: pilot explanation for Fig. 2 search episode is ((SS1) (SS2, SS3) (SS4))

Figure 3. Data structures used in the EGBG Algorithm

3. Having the pilot explanation available permits search at each successive level to be conducted by a variety of heuristics. That is, the search is no longer restricted to the 'top-down' serial processing dictated by standard Graphplan backward search. As long as all the constraints within the PE (as described above) are addressed before extending the planning graph, they can be processed in any desired order. For example, the approach employed in our initial experiments is to conduct search from the bottom-up during each search episode. A promising idea is to first check the PE backtrack points that lie closest to the initial state (level 0), which is the goal of the backward search. If they can be extended down to level 0 considerable search effort during that episode will be avoided.

There are of course costs incurred in building and using this potentially large search trace structure. These are addressed in section 3.3 after we first describe the algorithm for using the pilot explanation to direct all future search and planning graph extension.

3.2 Using the Pilot Explanation

EGBG (Explanation-Guided Backward-search for Graphplan) can be viewed as consisting of 3 phases. The two preliminary phases that enable the use of the PE to speed up search can be summarized as:

1. *Initial Planning graph Build Ph*ase -From the initial state build & extend the planning graph in standard Graphplan fashion until a level is reached containing the goals: separately identify/mark actions that are static and dynamic mutex
2. *First Backward Search* -Conduct the 1st backward search from the top level with goals in standard Graphplan fashion. But for *each* 'search segment', SS (see Fig. 3 for the definition of search segment):
   - Save in SS, the goals addressed in that segment
   - Save in SS, in the order encountered and grouped by goal: an identifier for results of each action assignment attempted
   - When all segment goals have been satisfied add the list of all actions assigned & a pointer to *current* search segment to the next lower level search segment
   - Insert each completed search segment into the 'pilot explanation'. If backward search reaches the initial state, return plan; else extend the planning graph one level.

When backward search fails to find a solution to satisfy the plan goals the pilot explanation acts as an explanation for this failure at the given planning graph level. An alternate view is that the PE provides an explicit guide for all future search that must be conducted (to ensure soundness) at the next planning graph level.

Let's take the trace in Fig. 2 as an example PE produced after the first attempt at backward search on the problem goals WXYZ at level k. Since a solution was not found (as evidenced by the fact that the trace does not extend down to the initial state), the planning graph is extended to level k+1 and from this point on the pilot explanation directs all search for the problem solution. There are two processes entailed that can be interleaved in *any order:*

1. Visit and check each of the backtrack points in the PE (hereafter termed 'checkpoints')
2. Resume backward search wherever a checkpoint no longer holds.

Verifying Checkpoints

Note that the trace in Fig. 2 can be viewed as four intra-level search segments; there is one search segment under level k, two under level k-1 and one under level k-2. All checkpoint verification must be done in the context of the current planning graph level that corresponds to the pilot explanation level the checkpoints appear in. Once all checkpoints in a given search segment are processed, the subgoals that were the targets of that partial search can be memoized at that level, because (assuming a solution wasn't reached) they necessarily comprise a nogood. This corresponds to the memoization performed by standard Graphplan when it backtracks out of a level during the search process.

In Fig. 2 there are 15 action mutex checkpoints and 6 nogood checkpoints to be processed. Eleven of these entail no significant processing at all:

- 5 of the mutex checkpoints are due to static mutex conditions between actions –that is the last assigned action clobbers either an effect or precondition of some action assigned higher up in that level. Such mutex conditions can never be relaxed so there is no need to check their validity on each planning graph extension. Although the relative percentages vary by problem domain, generally 20 – 50% of all mutex conditions encountered during the course of a problem search are of the static type.
- Six of the checkpoints are associated with the nogood checking that occurs when all goals in a search level are successfully assigned and the resulting subgoals are compared against the memoized nogoods for the next level. These constitute another type of redundant checking performed by Graphplan that can be avoided by using the PE. The successful transfers of subgoals to the next level in the PE (the clear arrows in Fig. 2) are guaranteed to be valid at all future levels -i.e. they

```
SEARCH-WITH-PILOT-EXPLANATION ( PE: pilot
                                explanation, PG: plan graph)
L1: Let n = number of proposition levels in plan graph
    For p = [length of pilot explanation] to 0
        (i.e. from deepest level searched previously to top level)
    For each search segment SS in level p of the pilot explanation
      Store the goal-by-goal results contained in SS-assigns in
          Gresults and clear SS-assigns
      Call Process-Assign-Level (nil, SS-goals, SS, Gresults,
                                 PG, n-p )
      If no plan found, memoize SS-goals at plan graph level n-p
PE processing complete, no soln found for n-level plan graph.
   -extend the plan graph one level
   -translate the PE up to the new top level
Go to L1

PROCESS-ASSIGN-LEVEL (A: actions already assigned,
  G: goals left to assign, SS: search segment, Gresults: unproc-
  essed assign results, PG: plan graph, k: pg level )
If G is not empty
  Select front goal g from G
  Let Ag be the set of actions from level k in PG that support g
  Add empty Newresults list to end of SS-assigns (will hold the
      results of action assigns for g)
  Select from Gresults the front set of assignment results:
    gresults (from previous search episode)
L1: Select front action act in Ag
    If Gresults is empty then act and rest of Ag are new
      establishers of g at level k:
      Call PROCESS-NEW-ACTIONS (A, G, Ag, Newresults,
                                SS, PG, k )
    else select front assign result, ares, from gresults
      If ares = 'ok' action has no conflicts, --move to next goal:
       Call PROCESS-ASSIGN-LEVEL ((A + act), (rest G), SS,
                                  Gresults, PG, k)
      else if ares is an integer, act caused a dynamic mutex
          checkpoint, --it must be tested:
          If act is still dyn mutex wrt the action at position
            ares of A
            Add ares to end of Newresults
          else mutex no longer holds, resume search & extend PE
            Call ASSIGN-GOALS ((rest G), (A + act), SS, PG, k)
            (If backward search reaches the init state, returns plan)
            Add 'ok' to end of Newresults
       else ares is a checkpoint that does not need to be tested
          Add ares to end of Newresults
  If Ag is empty return, else go to L1

PROCESS-NEW-ACTIONS (A: actions already assigned,
  G: goals to assign, N: new actions for front goal,
  Newresults: list of new assign results for front goal of G,
  SS: search segment, PG: plan graph, k: pg level )
Let g be the front goal in G
L1: If N is empty, return
    else Select front action act from N
        If act is mutex with some action in A
          save the appropriate assign result ('nt' for stat mutex,
            integer for dyn mutex) in Newresults
          go to L1
        else assign act to g and resume backward search on
          remaining goals (extending the PE):
            Call ASSIGN-GOALS ((rest G), (A + act ), SS, PG, k )
            (if backward search reaches the init state, returns plan)
        go to L1.
```

Figure 4. Pseudo code description of EGBG's 3rd phase

cannot become a nogood at a higher planning graph level. The specific subgoals found to constitute a nogood (indicated by black arrows in Fig. 2) may, however, no longer be in conflict at some higher plan graph level that the PE is translated to. But since *all* search segments above and below the nogood of concern will necessarily be processed before this search episode ends the nogoods will just as necessarily be "proven" in the process -or a solution will be found making the issue moot.

Resuming Backward Search

We next describe the process entailed when a checkpoint is found to no longer hold at a new level. Amongst the checkpoints then, only dynamic mutex checkpoints must be tested. If such a mutex is found to no longer hold, the checkpoint indicates a position at which backward search must be resumed.

Invalid Mutex Conditions: If an action mutex checkpoint in the explanation is found to no longer hold at level k+1, this is indicative of a possible means of satisfying the goals that was disallowed at the previous level. Backward search therefore resumes *at the point where the mutex caused backtracking*. The action assignments for the goal propositions up to the point of the mutex failure are augmented with the action assignment that was precluded by the (no longer valid) mutex. Search continues below this point until either a solution is found or the resumption point is returned to. The PE trace is extended during this search and the mutex checkpoint is replaced by the subtrace so generated.

To ensure that the pilot explanation is complete in it's encapsulation of a Graphplan backward search episode we must also check for the presence of new actions that may be establishers of a goal at level k but not at level k-1 when the PE trace was built.

New Actions: A goal proposition 'P' at level k+1 could have more establishing actions than at level k, indicating that a new action has been added during planning graph extension. At each occurrence of unassigned proposition P in the pilot explanation search is resumed using the values of any previously assigned goal props and augmenting them with the new action assignment to P. Search continues until either a solution is found or the resumption point is returned to. In the latter case the trace for this search segment is inserted via a new establisher of P in the PE.

3.3 Costs and Tradeoffs

The overhead entailed in building the pilot explanation, storing adequate information in it, and subsequently using it is not insignificant. It's comprised primarily of:

1. PE construction time cost during backward search
2. Startup cost for backward search resumption –incurred each time a checkpoint no longer holds and the information needed to re-initiate search must be gathered.
3. Storage space required to retain the PE.

The first two of these factors will work directly to offset the three speedup factors discussed in section 3.1 The last factor impacts not only the machine storage requirements but, depending on a given platform's "garbage collection" scheme, can have a significant impact on runtime also.

For "small" problems (where the solution is found quickly by Graphplan after only a few search episodes) this approach can be expected to show little if any advantage. The overhead of constructing the pilot explanation is likely to more than offset the few search episodes in which it could be used to shortcut the search. The real speedup is likely to be seen in the larger problems involving many episodes of deep search on the planning graph. However, as we discuss in the

Problem	EGBG				Standard Graphplan			Speedup
	Total Time	Bktrks	Mutex Cheks	Size of PE	Total Time	Bktrks	Mutex Cheks	
BW-Large-B (18/18)	4.13	1100 K	3566 K	944	9.8	2823 K	8117 K	2.4x
Rocket-ext-a (7/36)	5.05	906 K	215 K	320	53.2	8128 K	2944 K	10.5x
Tower-5 (31/31)	.9	316 K	643 K	2722	21.4	7907 K	23040 K	23.8x

Table 1. Comparison of EGBG with standard Graphplan. Times given in cpu minutes on a sparc ultra 1 running Allegro common lisp. Numbers in parentheses next to the problem names list the number of time steps and number of actions respectively in the solution. A measure of the backtracks and mutex checks performed during the search are also shown. "Size of PE" gives a measure of the pilot explanation size in terms of the final number of "search segments"

Problem	EGBG with EBL/DDB				EGBG				Speed up
	Total Time	Bktrks	Mutex Cheks	Size of PE	Total Time	Bktrks	Mutex Cheks	Size of PE	
BW-Large-A (12/12)	11.9 s	2.9 K	1.1 K	47	12.5 s	3.5 K	1.4 K	75	1.05x
Rocket-3-2-5 (5/20)	12.7 s	2.0 K	1.0 K	26	24.4 s	2.0 K	1.0 K	66	1.92x
Hanoi-Tower-3	23.3 s	14.1 K	66.7 K	170	52.2 s	38.3 K	300 K	283	2.24x

Table 2. Comparison of EGBG and EGBG augmented with EBL\ DDB on 3 small problems . Times are given in cpu seconds. A measure of the backtracks and mutex checks performed during the search are also shown. "Size of PE" gives a measure of the pilot explanation size in terms of the number of "search segments" it contains in the end.

next section, these are also the kinds of problems that can generate huge pilot explanations with all the storage and garbage collection problems it entails.

4. Experimentation

The EGBG program involved a major augmentation and rewrite of a Graphplan implementation in Lisp[1]. The planning graph extension routines were augmented to discriminate between static and dynamic mutexes and to facilitate the translation of the PE across levels. The backward search routines were augmented to build the PE during search. Once the initial PE is built, search control is taken over by a separate set of routines that process any dynamic checkpoints and determine when search must be resumed.

Table 1 provides a comparison of EGBG with standard Graphplan on some "benchmark" size problems from [Kautz & Selman, 96] in three domains. The speedup advantage of EGBG is respectable on these problems, ranging from 2.4 to almost 24 times faster.

The experience to date with the EGBG system on some of the larger problems has shown it to be sensitive to space and memory management issues. While generally conducting it's level search significantly faster than Graphplan in the early to middle stages of the problem search, EGBG may eventually gets bound up by the Lisp garbage collection system before it can complete it's solution search. For example the Ferry-41 problem is solved by standard Graphplan at level 27 of the planning graph in approximately 123 minutes[2]. Up to level 18, EGBG search is 25 times faster than standard Graphplan search. However around level 19 the current implementation of EGBG succumbs to the (as yet non-optimized) garbage collection system and doesn't emerge!

There are a variety of approaches that can be pursued to enable EGBG to extend its reach to the higher levels and large search spaces. The next section explores the role of the size of the PE.

5. Minimizing the Pilot Explanation

Any reduction that can be made to the size of the pilot explanation pays off in two important ways:

- There are fewer checkpoints that need to be processed at each planning graph extension
- The storage (and garbage collection) requirements are reduced.

The blind search conducted by Graphplan travels down many fruitless search branches that are essentially redundant to other areas of its search trace. And of course EGBG builds these into the pilot explanation and retains key aspects of them, revisiting dynamic checkpoints at their leaf nodes during each backward search episode.

We are investigating two different approaches to minimizing the PE so as to extend the range of problems addressable with EGBG under our current machine limitations.

One approach that immediately presented itself was to take advantage of concurrent work being conducted in our group on improving Graphplan's memoization routines by adding dependency-directed backtracking (DDB) and explanation-based learning (EBL) capabilities [Kambhampati, 99]. The EBL and DDB strategies complement our explanation-guided backward search technique quite naturally. The presence of EBL and DDB techniques reduces the sizes of the PE's and also makes them more likely to be directly applicable in future levels (as the dynamic checkpoints embodied in the explanation are more focused). At the same time, the PE provides a more powerful way of ex-

[1] The original Lisp implementation of Graphplan was done by Mark Peot and subsequently improved by David Smith

[2] On an Ultra-Sparc, compiled for speed, with improved memoization based on the "UB-Tree" structures developed by Koehler and her co-workers [Koehler et. al., 97]

ploiting the past searches than EBL-based nogoods alone do.

The integration of the EBL and DDB strategies into our EGBG system significantly changes the manner in which the pilot explanation is used during each search episode. Rather than visiting and processing each checkpoint in the explanation the conflict sets generated by the mutex checks and those returned by any partial search conducted can be used to indicate *precisely which checkpoints* should be processed. This avoids visiting and possibly conducting search below checkpoints that ultimately have no possibility of leading to a solution.

Although incorporating EBL/DDB into the backward search routines in EGBG is straightforward, it is more problematic to implement efficient processing of conflict sets in the routines that search using the pilot explanation. Table 2 shows the results produced by an EBL/DDB enhanced version of EGBG on the 3 such problems. The speedup improvement is modest, as expected, since the time spent in search for these small problems is only a fraction of the total time. However, the table also shows the dramatic impact that EBL/DDB has on the size of the pilot explanation, reducing it by roughly 50%. We are currently rewriting this part of EGBG as our first version turned out to have some implementation inefficiencies that inhibited it from scaling up to larger problems.

A second promising approach is to restrict the size of the pilot explanation. EGBG currently "grows" the PE every time backward search is resumed, without regard to it's size or the possible relevance of the "explanation" it is building. However, the PE can easily be made static after a given size limit is reached and still provide, during each backward search episode, all the search shortcuts it embodies. Alternately, some type of relevance-based heuristic could be employed to determine what portions of a learned PE are mostly likely to be useful and hence should be retained. That is, the well-known EBL issue of 'how much is too much' in the way of rules (or nogoods) also applies to EGBG and it's use of the pilot explanation.

6. Related Work

Verfaillie and Schiex [94] discuss and demonstrate various EBL-type methods for improving dynamic constraint satisfaction problem (DCSP) solving, which underlies the Graphplan backward search. Our approaches, while related, are much better customized to take advantage of the particular type of DCSP structural symmetry in Graphplan backward search. As we noted earlier, our approach is also related to, and complements, the approaches to add EBL and DDB capabilities to Graphplan. In particular, EBL approaches can only help decide if all methods of making a goal-set true at a particular level are doomed to fail. The pilot explanation, on the other hand, also guides the search away from branches likely to fail, and allows the planner to capitalize on the search penetration done at previous levels. In this sense, the pilot explanations are also related to the “sticky values” approach for improving CSP [Frost & Dechter, 94]

7. Conclusion and Future Directions

We presented a method for exploiting the symmetry in the planning graph structure of Graphplan algorithm to improve its backward search. The main insight underlying our method is that due to the inherent symmetry of the planning graph the backward search conducted at level k + 1 of the graph is essentially a replay of the search conducted at the previous level k with certain well-defined extensions. We presented a structure called the pilot explanation, which captures the failures encountered at previous levels of the search. This structure is used in an intelligent way to guide the search at the newer levels. We implemented this approach in a system called EGBG, and presented a preliminary empirical study with the system. We also discussed the importance of minimizing the size of the pilot explanation and some relevant techniques, including a recent EBL and DDB implementation for Graphplan. Our near-term focus will be to address the problems currently limiting the size of problems that can be handled by the system. Future areas of investigation will include a study of various heuristics and methods for using the PE to direct search. These could entail jumping about to process the various checkpoints in a "best first" manner or even deliberately not processing certain unpromising checkpoints (and thereby accepting the loss of soundness) in the hopes of short-cutting the search.

References

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)*)

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

Ihrig, L, and Kambhampati, S. 1997. Storing and Indexing Plan Derivations through Explanation-based Analysis of Retrieval Failures. *Journal of Artificial Intelligence.* Vol. 7. 1997.

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

Kambhampati, S. 1998. On the relations between Intelligent Backtracking and Failure-driven Explanation Based Learning in Constraint Satisfaction and Planning. *Artificial Intelligence. Vol 105.*

Kambhampati, S. 1999. Improving Graphplan's search with EBL & DDB. In *Proc. IJCAI-99.* 1999.

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

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

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

Schiex, T, and G.Verfaillie. Nogood Recording for Static and Dnamic Constraint Satisfaction Problems. C.E.R.T-O.N.E.R.A.

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

Verfaillie, G and Schiex, T. 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. In Proc. AAAI.

Veloso, M. 1994. Flexible strategy learning: Analogical replay of problem solving episodes. In Proc. AAAI.