Journal of Artificial Intelligence Research 23 (2005) 533-585 Submitted 4/04; published 5/05

Using Memory to Transform Search on the Planning Graph

Terry Zimmerman WIZIM@CS.CMU.EDU
Robotics Institute, Carnegie Mellon University
Pittsburgh, PA 15213-3890

Subbarao Kambhampati RAO@ASU.EDU
Department of Computer Science & Engineering
Arizona State University, Tempe AZ 85287-5406

Abstract

The Graphplan algorithm for generating optimal make-span plans containing parallel sets of actions remains one of the most effective ways to generate such plans. However, despite enhancements on a range of fronts, the approach is currently dominated in terms of speed, by state space planners that employ distance-based heuristics to quickly generate serial plans. We report on a family of strategies that employ available memory to construct a search trace so as to learn from various aspects of Graphplan's iterative search episodes in order to expedite search in subsequent episodes. The planning approaches can be partitioned into two classes according to the type and extent of search experience captured in the trace. The planners using the more aggressive tracing method are able to avoid much of Graphplan's redundant search effort, while planners in the second class trade off this aspect in favor of a much higher degree of freedom than Graphplan in traversing the space of 'states' generated during regression search on the planning graph. The tactic favored by the second approach, exploiting the search trace to transform the depth-first, IDA* nature of Graphplan's search into an iterative state space view, is shown to be the more powerful. We demonstrate that distance-based, state space heuristics can be adapted to informed traversal of the search trace used by the second class of planners and develop an augmentation targeted specifically at planning graph search. Guided by such a heuristic, the step-optimal version of the planner in this class clearly dominates even a highly enhanced version of Graphplan. By adopting beam search on the search trace we then show that virtually optimal parallel plans can be generated at speeds quite competitive with a modern heuristic state space planner.

1. Introduction

When Graphplan was introduced in 1995 (Blum & Furst, 1995) it became one of the fastest programs for solving the benchmark planning problems of that time and, by most accounts, constituted a radically different approach to automated planning. Despite the recent dominance of heuristic state-search planners over Graphplan-style planners, the Graphplan approach is still one of the most effective ways to generate the so-called "optimal parallel plans". State-space planners are drowned by the exponential branching factors of the search space of parallel plans (the exponential branching is a result of the fact that the planner needs to consider each subset of non-interfering actions). Over the 8 years since its introduction, the Graphplan system has been enhanced on numerous fronts, ranging from planning graph construction efficiencies that reduce both its size and build time by one or more orders of magnitude (Smith & Weld, 1998; Long & Fox, 1999), to search speedup techniques such as variable and value ordering, dependency-directed backtracking, and explanation based learning (Kambhampati, 2000). In spite of these advances, Graphplan has ceded the lead in planning speed to a variety of heuristic-guided planners (Bonet & Geffner, 1999; Nguyen & Kambhampati, 2000; Gerevini & Serina, 2002). Notably, several of these exploit the planning graph for powerful state-space heuristics, while

eschewing search on the graph itself. Nonetheless, the Graphplan approach remains perhaps the fastest in parallel planning mainly because of the way it combines an iterative deepening A* ("IDA*", Korf, 1985) search style with a highly efficient CSP-based incremental generation of applicable action subsets.

We investigate here the use of available memory so as to surmount some of Graphplan's major drawbacks, such as redundant search effort and the need to exhaustively search a k-length planning graph before proceeding to the k+1 length graph. At the same time we wish to retain attractive features of Graphplan's IDA* search such as rapid generation of parallel action steps and the ability to find step optimal plans. The approach we describe remains rooted in iterative search on the planning graph but greatly expedites this search by building and maintaining a concise search trace.

Graphplan alternates between two phases; one in which a data structure called a "planning graph" is incrementally extended, and a backward phase where the planning graph is searched to extract a valid plan. After the first regression search phase the space explored in any given episode is closely correlated with that conducted in the preceding episode. The strategy we pursue in this work is to employ an appropriately designed trace of the search conducted in episode n (which failed to find a solution) to identify and avoid those aspects of the search that are provably unchanged in episode n+1, and focus effort on features that may have evolved. We have identified precisely which features are dynamic across Graphplan search episodes and construct search traces that capture and exploit these features to different degrees. Depending on its design a search trace may provide benefits such as 1) avoidance of much of Graphplan's redundant search effort, 2) learning from its iterative search experience so as to improve its heuristics and the constraints embodied in the planning graph, and 3) realizing a much higher degree of freedom than Graphplan, in traversing the space of 'states' generated during the regression search process. We will show that the third advantage is particularly key to search trace effectiveness, as it allows the planner to focus its attention on the most promising areas of the search space.

The issue of how much memory is the 'right' amount to use to boost an algorithm's performance cuts across a range of computational approaches from search to the paging process in operating systems, and Internet browsing to database processing operations. In our investigation we explore several alternative search trace based methods that differ markedly in terms of memory demands. We describe four of these approaches in this paper. Figure 1 depicts the pedigree of this family of search trace-based planners, as well as the primary impetus leading to the evolution of each system from its predecessor. The figure also suggests the relative degree to which each planner steps away from the original IDA* search process underlying Graphplan. The two tracks correspond to two genres of search trace that we have developed;

- *left track:* The EGBG planners (Explanation Guided Backward search for Graphplan) employ a more comprehensive search trace focused on minimizing redundant search.
- *right track:* The PEGG planners (Pilot Explanation Guided Graphplan) use a more skeletal trace, incurring more of Graphplan's redundant search effort in exchange for reduced memory demands and increased ability to exploit the state space view of the search space.

The EGBG planner (Zimmerman & Kambhampati, 1999) adopts a memory intensive structure for the search trace as it seeks primarily to minimize redundant consistency-checking across Graphplan's search iterations. This proves to be effective in a range of smaller problems but memory constraints

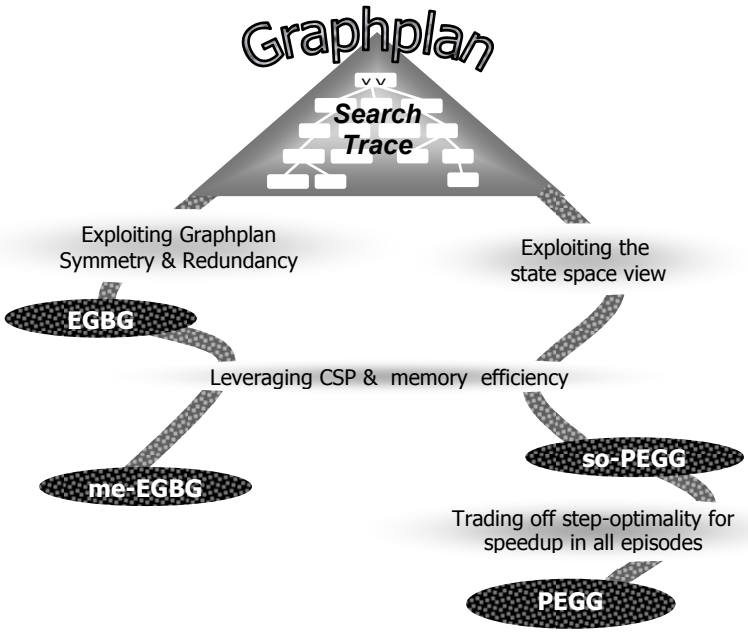

Figure 1: Applying available memory to step away from the Graphplan search process; A family of search trace-based planners

impede its ability to scale up. Noting that Graphplan's search process can be viewed as a specialized form of CSP search (Kambhampati, 2000), we explore some middle ground in terms of memory usage by augmenting EGBG with several methods known to be effective as *speedup* techniques for CSP problems.

Our primary interest in these techniques, however, is the impact on memory reduction and we describe how they accomplish this above and beyond any search speedup benefit they afford. The implemented planner, me-EGBG, markedly outperforms EGBG in speed and capabilities, but a variety of problems still lie beyond the planner's reach due to memory constraints.

The search trace structure used by the PEGG track planners trades off minimization of redundant search in exchange for a much smaller memory footprint. In addition to its greatly reduced memory demands, the PEGG search trace structure can be exploited for its intrinsic state space view of what is essentially Graphplan's CSP-oriented search space. A significant speedup advantage of this approach over Graphplan and the EGBG track planners derives from its ability to employ the 'distance-based' heuristics that power many of the current generation of state-space planners (Bonet & Geffner, 1999; Nguyen & Kambhampati, 2000; Hoffman, 2001). We adapt these heuristics to the task of identifying the most promising states to visit in the search trace and implement the approach first in the so-PEGG planner ('step-optimal PEGG', Zimmerman & Kambhampati, 2003). So-PEGG outperforms even a highly enhanced version of Graphplan by up to two orders of magnitude in terms of speed, and does so while maintaining the guarantee of finding a step-optimal plan.

Finally we explore adoption of a beam search approach in visiting the state space implicit in the PEGG-style trace. Here we employ the distance-based heuristics extracted from the planning graph itself, not only to direct the order in which search trace states are visited, but also to prune and restrict that space to only the heuristically best set of states, according to a user-specified metric. We show that the planning graph can be further leveraged to provide a measure of the likelihood that a previously generated regression state might spawn new search branches at a higher planning graph level.

We term this metric 'flux' and employ it in an effective filter for states that can be skipped over even though they might appear promising based on the distance-based heuristic. Implemented in the PEGG system (Zimmerman & Kambhampati, 2003), this approach to exploiting a search trace produces a two-fold benefit over our previous approaches; 1) further reduction in search trace memory demands and 2) effective release from Graphplan's exhaustive search of the planning graph in *all* search episodes. PEGG exhibits speedups ranging to more than 300x over the enhanced version of Graphplan and is quite competitive with a recent state space planner using similar heuristics. In adopting beam search PEGG necessarily sacrifices the *guarantee* of step-optimality but empirical evidence indicates the secondary heuristics are remarkably effective in ensuring the make-span of solutions produced are virtually at the optimal.

The fact that these systems successfully employ a search trace *at all* is noteworthy. In general, the tactic of adopting a search trace for algorithms that explicitly generate node-states during iterative search episodes, has been found to be infeasible due to memory demands that are exponential in the depth of the solution. In Sections 2 and 3 we describe how tight integration of the search trace with the planning graph permits the EGBG and PEGG planners to largely circumvent this issue. The planning graph structure itself can be costly to construct, in terms of both memory and time; there are well-known problems and even domains that are problematic for planners that employ it. (Post-Graphplan planners that employ the planning graph for some purpose include "STAN", Long & Fox, 1999, "Blackbox", Kautz & Selman, 1999, "IPP", Koehler et al., 1997, "AltAlt", Nguyen & Kambhampati, 2000, "LPG" Gerevini & Serina, 2002). The planning systems described here share that memory overhead of course, but interestingly, we have found that search trace memory demands for the PEGG class of planners have not significantly limited the range of problems they can solve.

The remainder of the paper is organized as follows: Section 2 provides a brief overview of the planning graph and Graphplan's search process. The discussion of both its CSP nature and the manner in which the process can be viewed as IDA* search motivates the potential for employing available memory to accelerate solution extraction. Section 3 addresses the two primary challenges in attempting to build and use a search trace to advantage with Graphplan: 1) How can this be done within reasonable memory constraints given Graphplan's CSP-style search on the planning graph? and, 2) Once the trace is available, how can it most effectively be used? This section briefly describes EGBG (Zimmerman & Kambhampati, 1999), the first system to use such a search trace to guide Graphplan's search, and outlines the limitations of that method (Details of the algorithm are contained in Appendix A.) Section 4 summarizes our investigations into a variety of memory reduction techniques and reports the impact of a combination of six of them on the performance of EGBG. The PEGG planners are discussed in Section 5 and the performance of so-PEGG and PEGG (using beam search) are compared to an enhanced version of Graphplan, EGBG, and a modern, serial state-space planner. Section 6 contains a discussion of our findings and Section 7 compares this work to related research. Finally, Section 8 wraps up with our conclusions.

2. Background & Motivation: Planning Graphs and the Nature of Direct Graph Search

Here we outline the Graphplan algorithm and discuss traits suggesting that judicious use of additional memory might greatly improve its performance. We touch on three related views of Graphplan's search; 1) as a form of CSP, 2) as IDA* search and, 3) its state space aspect.

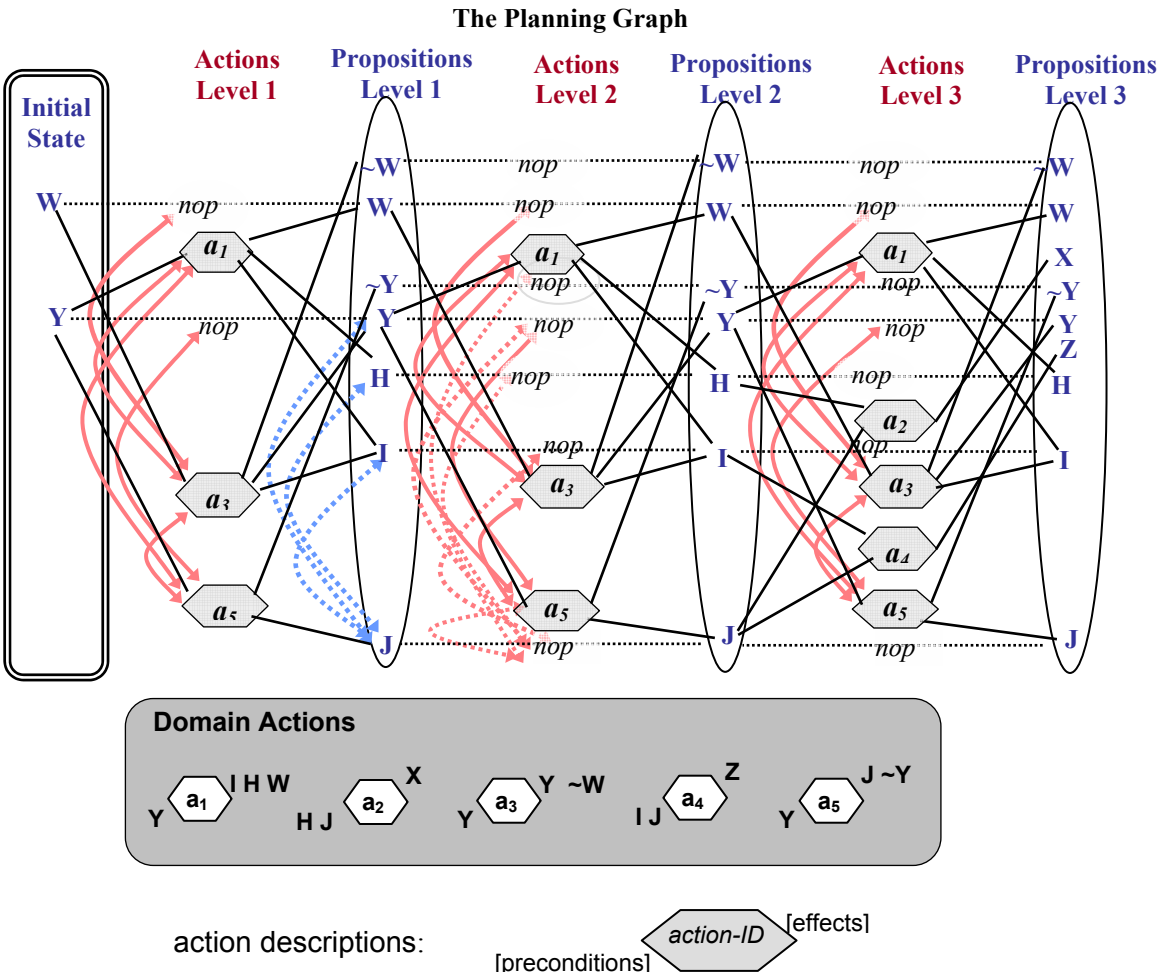

Figure 2: Planning graph representation for three levels in the *Alpha* domain

2.1 Construction and Search on a Planning Graph

The Graphplan algorithm employs two interleaved phases – a forward phase, where a data structure called a "planning graph" is incrementally extended, and a backward phase where the planning graph is searched to extract a valid plan. The planning graph consists of two alternating structures, called proposition lists and action lists. At the bottom of Figure 2 is depicted a simple domain we will refer to as the *Alpha* domain and use for illustration in this study. The figure shows four action and proposition levels of the planning graph engendered by the simple initial state given the domain. We start with the initial state as the zeroth level proposition list. Given a k-level planning graph, the extension of the graph 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 of the domain model, "no operation" actions are introduced, one for each condition in the k^{th} level proposition list (abbreviated as "nop" in this paper's figures, but also termed "persists" by others). A "nop-C" action has C as its precondition and C as its effect. Given the k^{th} level actions, 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*, their preconditions in the level *k* proposition list, and their effects in the level *k+1* proposition list.

During planning graph construction binary "mutex" constraints are computed and propagated. In Figure 2, the arcs denote mutex relations between pairs of propositions and pairs of actions. The propagation starts at level 1 by labeling as mutex all pairs of actions that are statically interfering with each other ("static mutex"), that is their preconditions or effects are logically inconsistent. Mutexes are then propagated from this level forward using two simple propagation rules. 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 2 are then mutex if they are statically interfering or if a precondition of the first action is mutually exclusive with a precondition of the second . (We term the latter "dynamic mutex", since this constraint may relax at a higher planning graph level).[1] The propositions themselves can also be either static mutex (one negates the other) or dynamic mutex (all actions supporting one proposition are mutex with all actions supporting the other). To reduce Figure 2 clutter mutex arcs for propositions and their negations are omitted.

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. Figure 3 depicts Graphplan search in a manner similar to the CSP variable-value assignment process. Beginning with the propositions corresponding to the goals at level *k*, we incrementally select a set of actions from the level *k* action list that support all the goals, 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). This is essentially a CSP problem where the goal propositions at each level are the variables, actions that establish a proposition are the values, and the mutex conditions constitute constraints. The search proceeds in depth-first fashion: 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 (the initial state) and the solution is extracted by unwinding the recursive goal assignment calls. This process can be viewed as a system for solving "Dynamic CSPs" (DCSP) (Mittal & Falkenhainer, 1990; Kambhampati 2000), wherein the standard CSP formalism is augmented with the concept of variables that do not appear (a.k.a. get activated) until other variables are assigned.

During the interleaved planning graph extension and search phases, the graph may be extended to a stasis condition, after which no further changes occur in actions, propositions, or mutex conditions. A sufficient condition defining this "level-off" is a level where no new actions are introduced *and* no existing mutex conditions between propositions go away. We will refer to all planning graph levels at or above level-off as 'static levels'. Note that although the graph becomes static at this point, finding a solution may require many more episodes composed of adding identical static levels and conducting regression search on the problem goals.

Like many fielded CSP solvers, Graphplan's search process benefits from a simple form of nogood learning. 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. Subsequently, when the backward search process later enters level k with a set of subgoals they are first checked against the hash table, and if a match is found the search

[1] The static mutex condition has also been called "eternal mutex" and the dynamic mutex termed "conditional mutex" (Smith & Weld, 1998).

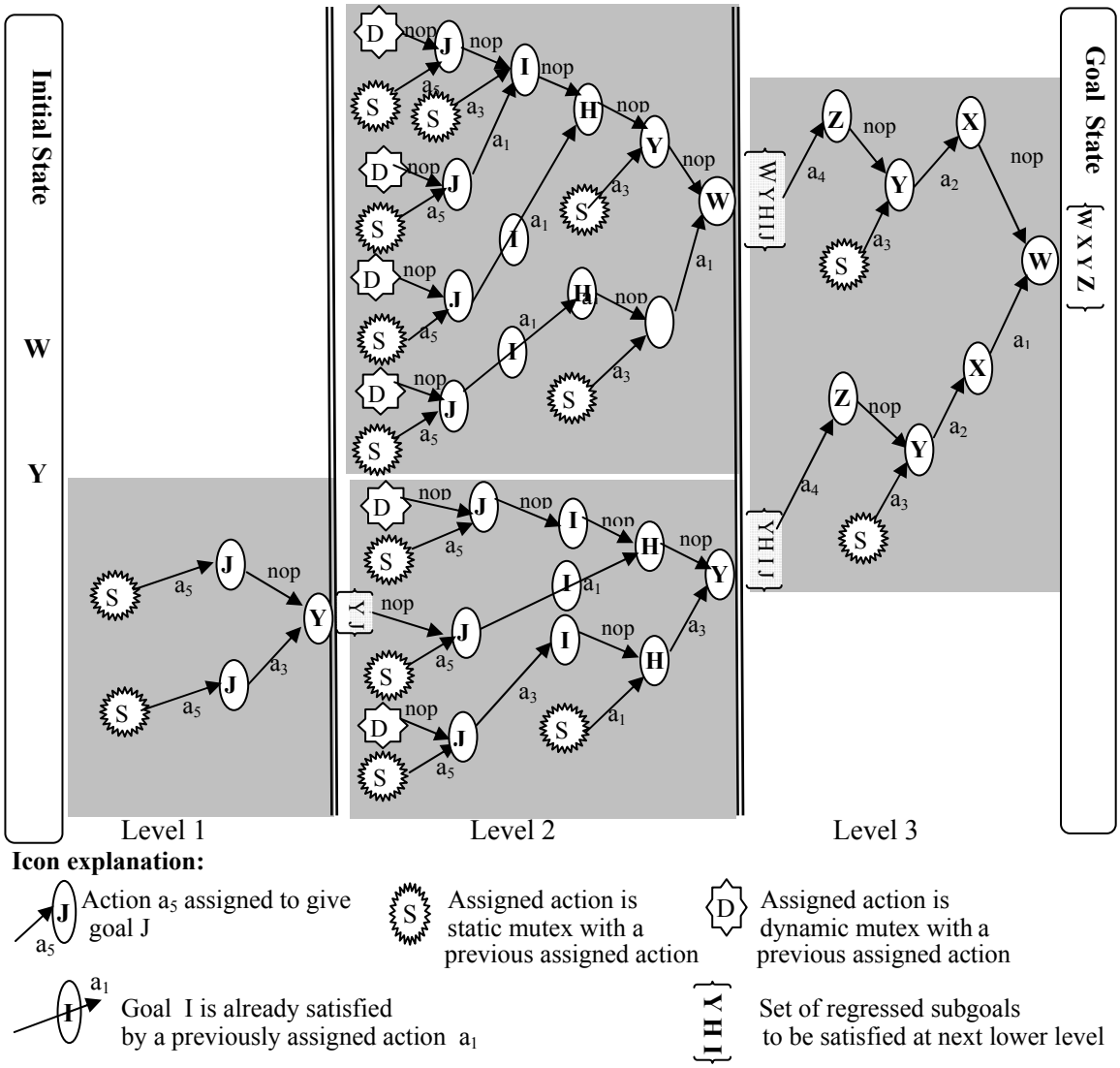

Figure 3: CSP-style trace of Graphplan's regression search on the Figure 2 planning graph

process backtracks. This constitutes one of three conditions for backtracking: the two others arise from attempts to assign static mutex actions and dynamic mutex actions (See the Figure 3 legend).

We next discuss Graphplan's search from a higher-level view that abstracts away its CSP nature.

2.2 Graphplan as State Space Search

From a more abstract perspective, Graphplan can be viewed as conducting regression state space search from the problem goals to the initial state. In this view, the 'states' that are generated and expanded are the subgoals that result when the CSP process for a given set of subgoals finds a consistent set of actions satisfying the subgoals at that planning graph level (c.f. Kambhampati & Sanchez, 2000). In this view the "state-generator function" is effectively Graphplan's CSP-style goal assignment routine that seeks a non-mutex *set of actions* for a given set of subgoals within a given planning graph level. This view is depicted in Figure 4, where the top graph casts the CSP-style search trace of

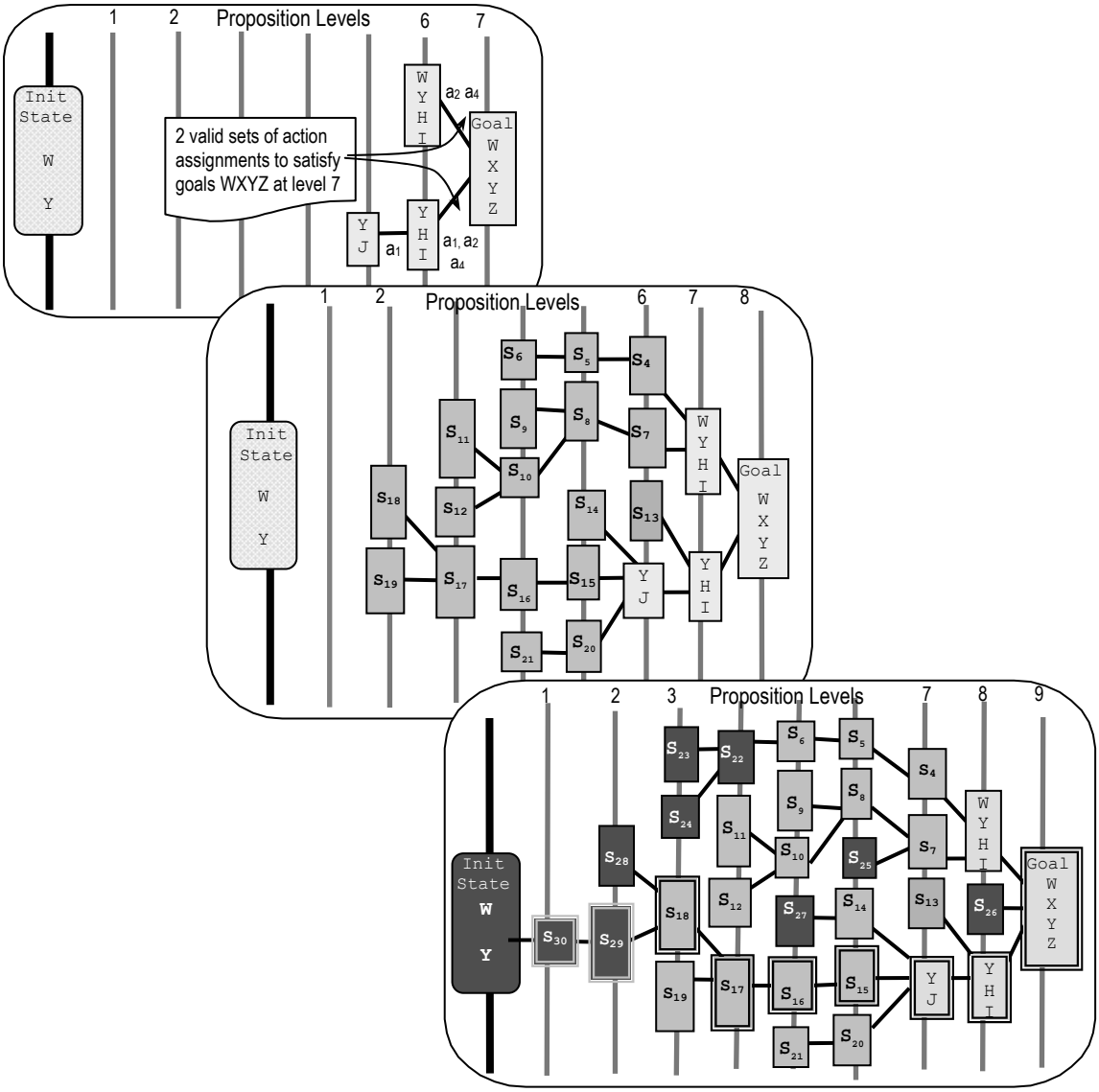

Figure 4: Graphplan's regression search space: Three consecutive search episodes

Figure 3 as a high-level state-space search trace. The terms in each box depict the set of (positive) subgoals that result from the action assignment process for the goals in the higher-level state to which the box is linked.[2]

Recognizing the state-space aspect of Graphplan's search helps in understanding its connection to IDA* search. First noted and briefly discussed in (Bonet & Geffner, 1999), we highlight and expand upon this relationship here. There are three correspondences between the algorithms:

1. Graphplan's episodic search process in which all nodes generated in the previous episode are re-generated in the new episode (and possibly some new nodes), corresponds to IDA*'s iterative search. Here the Graphplan nodes are the 'states' (sets of subgoals) that result when its regres-

[2] Figure 4 facilitates discussion of the search trace in the next section, by conjuring up a hypothetical problem in which the first search episode begins on level 7 of the planning graph instead of level 3, as in Figure 3.

sion search on a given plan graph level succeeds. From this perspective the "node-generator function" is effectively Graphplan's CSP-style goal assignment routine that seeks a non-mutex set of actions for a given set of propositions within a given planning graph level.

2. From the state space view of Graphplan's search (ala Figure 4), within a given search episode/iteration the algorithm conducts its search in the depth-first fashion of IDA*. This ensures that the space requirements are linear in the depth of a solution node.

3. The upper bound that is 'iteratively deepened' ala IDA* is the node-state heuristic f-value; f = g + h. In this context *h* is the distance in terms of associated planning graph levels between a state generated in Graphplan's regression search and the initial state[3] and *g* is the cost of reaching the state from the goal state in terms of number of CSP epochs (i.e. the numerical difference between the highest graph level and the state's level).

For our purposes, perhaps he most important observation is that the implicit f-value bound for a given iteration is just the length of the planning graph associated with that iteration. That is, for any node-state, its associated planning graph level determines both the distance to the initial state (h) and the cost to reach it from the goal state (g), and the total must always equal the length of the plan graph. This heuristic is clearly admissible; there can be no shorter distance to the goal because Graphplan exhaustively searches all shorter length planning graphs in (any) previous iterations. It is this heuristic implicit in the Graphplan algorithm which guarantees that a step-optimal solution is returned. Note that from this perspective *all nodes* visited in a given Graphplan search iteration *implicitly have the same f-value*: g + h = length of planning graph. We will consider implications of this property when we address informed traversal of Graphplan's search space in Section 5.

The primary shortcoming of a standard IDA* approach to search is that it regenerates so many of the same nodes in each of its iterations. It has long been recognized that IDA*'s difficulties in some problem spaces can be traced to using *too little* memory (Russell, 1992; Sen & Bagchi, 1989). The only information carried over from one iteration to the next is the upper bound on the f-value. Graphplan partially addresses this shortcoming with its memo caches that store "no-goods" -states found to be inconsistent in successive episodes. However, the IDA* nature of its search can make it an inefficient planner for problems in which the goal propositions appear non-mutex in the planning graph many levels before a valid plan can actually be extracted.

A second shortcoming of the IDA* nature of Graphplan's search is that all node-states generated in a given Graphplan episode have *the same f-value* (i.e. the length of the graph). As such, within an iteration (search episode) there is no discernible preference for visiting one state over another. We next discuss the use of available memory to target these shortcomings of Graphplan's search.

3. Efficient Use of a Search Trace to Guide Planning Graph Search

The search space Graphplan explores in a given search episode is defined and constrained by three factors: the problem goals, the plan graph associated with the episode, and the cache of memoized no-good states created in all previous search episodes. Typical of IDA* search there is considerable

[3] Bonet & Geffner define h_G (the Graphplan *h-value*) somewhat differently as the first level at which the goals of a state appear non-mutex and have not been memoized. Our definition (which is *not* necessarily the first level at which the *Sm* goals appear non-mutex) produces the most informed admissible estimate in all cases. This guarantees that all states generated by Graphplan have an f-value equal to the planning graph length, which is the property of primary interest to us.

similarity (i.e. redundancy) in the search space for successive episodes as the plan graph is extended. In fact, as discussed below, the backward search conducted at any level *k+1* of the graph is essentially a "replay" of the search conducted at the previous level *k* with certain well-defined extensions. More specifically, essentially *every* set of subgoals generated in the backward search of episode *n*, starting at level *k*, will be regenerated by Graphplan during episode *n+1* starting at level *k+1* (unless a solution is found first).[4]

Now returning to Figure 4 in its entirety, note that it depicts a state space tree structure corresponding to Graphplan's search over three consecutive iterations. The top graph, as discussed above, represents the subgoal 'states' generated in the course of Graphplan's first attempt to satisfy the WXYZ goal of a problem resembling our running example. (It is implied here that the W,X,Y,Z propositions are present in the planning graph at level 7 and that this is the *first* level at which no pair of them is mutex.) In the second search episode (the middle Figure 4 graph), the same states are generated again, but each at one level higher. In addition, these states are expanded to generate a number of children, shown in a darker shade. (Since Figure 4 is a hypothetical variation of the Alpha domain problem detailed in Figures 2 and 3, all states created beyond the first episode are labeled only with state numbers representing the order in which they are generated.) Finally, in the third episode, Graphplan regenerates the states from the previous two episodes in attempting to satisfy WXYZ at level 9, and ultimately finds a solution (the assigned actions associated with the figure's double outlined subgoal sets) after generating the states shown with darkest shading in the bottom graph of Figure 4.

Noting the extent to which consecutive iterations of Graphplan's search overlap, we investigate the application of additional memory to store a trace of the explored search tree. The first implemented approach, EGBG (which is summarized in the following subsection), seeks to leverage an appropriately designed search trace to avoid as much of the inter-episode redundant search effort as possible (Zimmerman & Kambhampati, 1999).

3.1 Aggressive Use of Memory in Tracing Search: The EGBG Planner

Like other types of CSP-based algorithms, Graphplan consumes most of its computational effort on a given problem in checking constraints. An instrumented version of the planner reveals that typically, 60 - 90% of the cpu run-time is spent in creating and checking action and proposition mutexes -both during planning graph construction and the search process. (Mutex relations incorporated in the planning graph are the primary 'constraints' in the CSP view of Graphplan, Kambhampati, 2000) As such, this is an obvious starting point when seeking efficiency improvements for this planner and is the primary tactic adopted by EGBG. We provide here only an overview of the approach, referring the interested reader to Appendix A for details.

EGBG exploits four features of the planning graph and Graphplan's search process:

- The set of actions that can establish a given proposition at level *k+1* is always a superset of those establishing the proposition at level *k*.

[4] Strictly speaking, this is not always the case due to the impact of Graphplan's memoizing process. For some problems a particular branch of the search tree generated in search episode *n* and rooted at planning graph level *k* may not be revisited in episode *n+1* at level *k+1* due to a 'no-good' proposition set memoized at level *k+1*. However, the memo merely acts to avoid some redundant search and neglecting these relatively rare cases serves to simplify visualization of the symmetry across Graphplan's search episodes. .

- The "constraints" (mutexes) that are active at level *k* monotonically decrease with increasing planning graph levels. That is, 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 it never gets re-activated at future 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.
- The problem goal set that is to be satisfied at a level *k* is the same set that will be searched on at level *k+1* when the planning graph is extended. That is, once a subgoal set is present at level *k* with no two propositions being mutex, it will remain so for all future levels.

Given an appropriate trace of the search conducted in episode *n* (which failed to find a solution) we would like to ignore those aspects of the search that are provably unchanged in episode *n*+1, and focus effort on only features that may have evolved. If previous search failed to extract a solution from the k-length planning graph, search on the *k+1* length graph can succeed only if one or more of the following conditions holds:

1. The dynamic mutex condition between some pair of actions whose concurrent assignment was attempted in episode *n* no longer holds in episode *n+1*.
2. For a subgoal that was generated in the regression search of episode *n* at planning graph level *k,* there is an action that establishes it in episode *n+1* and first appears in level *k+1*.
3. An episode *n* regression state (subgoal set) at level *k* that matched a cached memo at that level has no memo-match when it is generated at level *k+1* in episode *n+1*.

(The discussion in Appendix A formalizes these conditions.) In each instance where one of these conditions does *not* hold, a complete policy must resume backward search under the search parameters associated with the instance in the previous episode, *n*. Such resumed partial search episodes will either find a solution or generate additional trace subgoal sets to augment the parent trace. This specialized search trace can be used to direct *all* future backward search episodes for this problem, and can be viewed as an explanation for the failure of the search process in each episode. We hereafter use the terms *pilot explanation (PE)* and search trace interchangeably. The following definitions are useful in describing the search process:

Search segment: This is essentially a state, specifically a set of planning graph level-specific subgoals generated in regression search from the goal state (which is itself the first search segment). Each EGBG search segment S_n, generated at planning graph level *k* contains:

- A subgoal set of propositions to be satisfied
- A pointer to the parent search segment (S_p), (the state at level *k+1* that gave rise to S_n)
- A list of the actions that were assigned in S_p which resulted in the subgoals of S_n
- A pointer to the PE level (as defined below) associated with the S_n
- A sequential list of results of the action consistency-checking process during the attempt to satisfy S_n's subgoals. The possible trace results for a given consistency check are: static mutex, dynamic mutex, or action is consistent with all other prior assigned actions. Trace results are stored as a list of bit vectors for efficiency.

A search segment therefore represents a state plus some path information, but we often use 'search segment' and 'state' interchangeably. As such, all the boxes in Figure 4 (whether the state goals are explicitly shown or not) can be viewed as search segments.

Pilot explanation (PE): This is the search trace. It consists of the entire linked set of search segments representing the search space visited in a Graphplan backward search episode. It is convenient to visualize it as in Figure 4: a tiered structure with separate caches for segments associated with search on each planning graph level. We adopt the convention of numbering the PE levels in the *reverse order of the plan graph*: The top PE level is 0 (it contains a single search segment whose goals are the problem goals) and the level number is incremented as we move towards the initial state. When a solution is found, the PE will necessarily extend from the highest plan graph level to the initial state, as shown in the third graph of Figure 4.

PE transposition: When a state is first generated in search episode n it is associated with a specific planning graph level, say k. The premise of using the search trace to guide search in episode $n+1$ is based on the idea of re-associating each PE search segment (state) generated (or updated) in episode n with the next higher planning graph level. That is, we define *transposing* the PE as: For each search segment in the PE associated with a planning graph level k after search episode n, associate it with level $k+1$ for episode $n+1$.

Given these definitions, we note that the states in the PE after a search episode n on plan graph level k, loosely constitute the minimal set [5] of states that will be visited when backward search is conducted in episode $n+1$ at level $k+1$. (This bound can be visualized by sliding the fixed tree of search segments in the first graph of Figure 4 up one level.)

3.2 Conducting Search with the EGBG Search Trace

EGBG builds the initial pilot explanation during the first regression search episode while tracing the search process with an augmented version of Graphplan's "assign-goals" routine. If no solution is possible on the k-length planning graph, the PE is transposed up one level, and key features of its previous search are replayed such that significant new search effort only occurs at points where one of the three conditions described above holds. During any such new search process the PE is augmented according to the search space visited.

The EGBG search algorithm exploits its search trace in essentially bi-modal fashion: It alternates informed selection of a state from the search trace of its previous experience with a focused CSP-type search on the state's subgoals. Our discussion here of EGBG's bi-modal algorithm revolves around the second mode; minimizing redundant search effort once a state has been chosen for visitation. When we describe PEGG's use of the search trace in Section 5 we will see that greater potential for dramatic efficiency increases lies with the first mode; the selection of a promising state from the search trace.

After choosing a state to visit, EGBG uses the trace from the previous episode to focus on only those aspects of the entailed search that could possibly have changed. For each search segment S_i at planning graph level $k+1$, visitation is a 4–step process:

1. Perform a memo check to ensure the subgoals of S_i are valid at level $k+1$
2. 'Replay' the previous episode's action assignment sequence for all subgoals in S_i, using the segment's ordered trace vectors. Mutex checking is conducted on *only* those pairs of actions that were *dynamic* mutex at level k. For actions that are no longer dynamic mutex, add the can-

[5] It is possible for Graphplan's memoizing process to preclude some states from being regenerated in a subsequent episode. See footnote 2 for an brief explanation of conditions under which this may occur.

didate action to S_i's list of consistent assignments and resume Graphplan-style search on the remaining goals. S_i is augmented and the PE extended in the process. Whenever S_i's goals are successfully assigned, entailing a new set of subgoals to be satisfied at lower level *k,* a child search segment is created, linked to S_i, and added to the PE.

3. For each S_i subgoal in the replay sequence, check also for new actions appearing at level *k+1* that establish the subgoal. New actions that are inconsistent with a previously assigned action are logged as such in S_i's assignments. For new actions that do not conflict with those previously assigned, assign them and resume Graphplan-style search from that point as for step 2.

4. Memoize S_i's goals at level *k+1* if no solution is found via the search process of steps 2 and 3.

As long as *all* the segments in the PE are visited in this manner, the planner is guaranteed to find an optimal plan in the same search episode as Graphplan. Hereafter we refer to a PE search segment that is visited and extended via backward search to find a valid plan, as a *seed segment*. In addition, all segments that are part of the plan extracted from the PE we call *plan segments*. Thus, in the third graph of Figure 4, S_{18} is the apparent seed segment while the plan segments (in bottom up order) are; S_{30}, S_{29}, S_{18}, S_{17}, S_{16}, S_{15}, labeled segments YH, YHI, and the goal state WXYZ.

In principle we have the freedom to traverse the search states encapsulated in the PE in any order and are no longer restricted to the (non-informed) depth-first nature of Graphplan's search process. Unfortunately, EGBG incurs a high overhead associated with visiting the search segments in any order *other than* bottom up (in terms of PE levels). If an ancestor of any state represented in the PE were to be visited before the state itself, EGBG's search process would *regenerate* the state and any of its descendents (unless it first finds a solution). There is a non-trivial cost associated with generating the assignment trace information in each of EGBG's search segments; its search advantage lies in reusing that trace data without having to regenerate it.

On the other hand, top-down visitation of the segments in the PE levels is the degenerate mode. Such a search process essentially mimics Graphplan's, since each episode begins with search on the problem goal set, and (with the exception of the replay of the top-level search segment's assignments) regenerates all the states generated in the previous episode -plus possibly some new states- during its regression search. The search trace provides no significant advantage under a top-down visitation policy.

The bottom-up policy, on the other hand, has intuitive appeal since the lowest levels of the PE correspond to portions of the search space that lie closest to the initial state (in terms of plan steps). If a state in one of the lower levels can in fact be extended to a solution, the planner avoids all the search effort that Graphplan would expend in reaching the state from the top-level problem goals.

Adopting a bottom-up visitation policy amounts to layering a *secondary* heuristic on the primary IDA* heuristic, which is the planning graph length that is iteratively deepened. Recalling from Section 2.2 that *all* states in the PE have the same *f-value* in terms of the primary heuristic, we are essentially biasing here in favor of states with low *h-values.* Support for such a policy comes from work on heuristic guided state-space planning (Bonet & Geffner, 1999; Nguyen & Kambhampati, 2000) in which weighting *h* by a factor of 5 relative to the *g* component of the heuristic *f-value* generally improved performance. However, unlike these state-space planning systems, for which this is the primary heuristic, EGBG employs it as a secondary heuristic so the guarantee of step optimality does not

Problem	Standard Graphplan			EGBG				Speedup Ratios		
	Total Time	Backtracks	Mutex Checks	Total Time	Backtracks	Mutex Checks	Size of PE	Time	Bktrks	Mutex Chks
BW-Large-B (18/18)	213	2823 K	121,400 K	79	880 K	21,900 K	7919	2.7x	3.2x	5.5x
Rocket-ext-a (7/36)	402	8128 K	74,900 K	40	712 K	3,400 K	1020	10.0x	11.4x	22x
Tower-5 (31/31)	811	7907 K	23040 K	33	240 K	548 K	2722	24.5x	33x	42x
Ferry-6 (39/39)	319	5909 K	81000 K	62	977 K	8901 K	6611	5.1x	6.0x	9.1x

Table 1: Comparison of EGBG with standard Graphplan**.**
Numbers in parentheses give number of time steps / number of actions respectively. Search backtracks and mutex checks performed during the search are shown. "Size of PE" is pilot explanation size in terms of the final number of search segments. "Standard Graphplan" is the Lisp version by Smith and Peot.

depend on its admissibility. We have found bottom-up visitation to be the most efficient mode for EGBG and it is the default order for all EGBG results reported in this study.

3.3 EGBG Experimental Results

Table 1 shows some of the performance results reported for the first version of EGBG (Zimmerman & Kambhampati, 1999). Amongst the search trace designs we tried, this version is the most memory intensive and records the greatest extent of the search experience. Runtime, the number of search backtracks, and the number of search mutex checks performed is compared to the Lisp implementation of the original Graphplan algorithm. EGBG exhibits a clear advantage over Graphplan for this small set of problems;

- total problem runtime: 2.7 - 24.5x improvement
- Number of backtracks during search: 3.2 - 33x improvement
- Number of mutex checking operations during search: 5.5 - 42x improvement

Since total time is, of course, highly dependent on both the machine as well as the coding language [6] (EGBG performance is particularly sensitive to available memory), the backtrack and mutex checking metrics provide a better comparative measure of search efficiency. For Graphplan, mutex checking is by far the biggest consumer of computation time and, as such, the latter metric is perhaps the most complete indicator of search process improvements. Some of the problem-to-problem variation in EGBG's effectiveness can be attributed to the static/dynamic mutex ratio characterizing Graphplan's action assignment routine. The more action assignments rejected due to pair-wise statically mutex actions, the greater the advantage enjoyed by a system that doesn't need to retest them. Tower-of-Hanoi problems fall into this classification.

As noted in the original study (Zimmerman & Kambhampati, 1999) the range of problems that can

[6] All planners developed for this report were coded in Allegro Lisp and run on a Pentium 900 mhz, with 384 M RAM. Runtimes include plangraph construction time and exclude garbage collection time. Values in Table 1 differ from those published in 1999 because problems were re-run on this platform. They also reflect some changes in the tracking of statistics.

be handled by this implementation is significantly restricted by the amount of memory available to the program at runtime. For example, with a PE consisting of almost 8,000 search segments, the very modest sized BW-Large-B problem challenges the available memory limit on our test machine. We consider next an approach (*me-EGBG* in Figure 1) that occupies a middle ground in terms of memory demands amongst the search trace approaches we have investigated.

4. Engineering to Reduce EGBG Memory Requirements: The me-EGBG Planner

The memory demands associated with Graphplan's search process itself are not a major concern, since it conducts depth-first search with search space requirements linear in the depth of a solution node. Since we seek to avoid the redundancy inherent in the IDA* episodes of Graphplan's search by using a search trace, we must deal with a much different memory-demand profile. The search trace design employed by EGBG has memory requirements that are exponential in the depth of the solution. However, the search trace grows in direct proportion to the search space actually visited, so that techniques which prune search also act to greatly reduce its memory demands.

We examined a variety of methods with respect to this issue, and eventually implemented a suite of seven that *together* have proven instrumental in helping EGBG (and later, PEGG) overcome memory-bound limitations. Six of these are known techniques from the planning and CSP fields: variable ordering, value ordering, explanation based learning (EBL), dependency directed backtracking (DDB), domain preprocessing and invariant analysis, and transitioning to a bi-partite planning graph. Four of the six most effective methods are CSP *speedup* techniques, however our interest lies primarily in their impact on search trace memory demands. While there are challenging aspects to adapting these methods to the planning graph and search trace context, it is not the focus of this paper. Thus details on the motivation and implementation of these methods is relegated to Appendix B.

The seventh method, a novel variant of variable ordering we call 'EBL-based reordering', exploits the fact that we are using EBL *and* have a search trace available. Although the method is readily implemented in PEGG, the strict ordering of the trace vectors required by the EGBG search trace make it costly to implement for that planner. As such, 'memory-efficient EGBG' (me-EGBG) does not use EBL-based reordering and we defer further discussion until PEGG is introduced in Section 5.

4.1 Impact of Enhancements on EGBG Memory Demands

There are two major modes in which the first six techniques impact memory demand for me-EGBG: 1) Reduction in the size of the pilot explanation (search trace), either in the number of search segments (states), or the average trace content within the segments, and 2) Reduction in the requirements of structures that compete with the pilot explanation for available memory (i.e. the planning graph and the memo caches). Admittedly, these two dimensions are not independent, since the number of memos (though not the size) is linear in the number of search segments. We will nonetheless consider this partition in our discussion to facilitate the comparison of each method's impact on the search trace.

In general, the impact of each these enhancements on the search process depends significantly, not only on the particular problem, but also on the presence (or absence) of any of the other methods. No single configuration of techniques proves to be optimal across a wide range of problems. Indeed, due to computational overhead associated with these methods, it is generally possible to find a class of problems for which planner performance *degrades* due to the presence of the method. We chose this

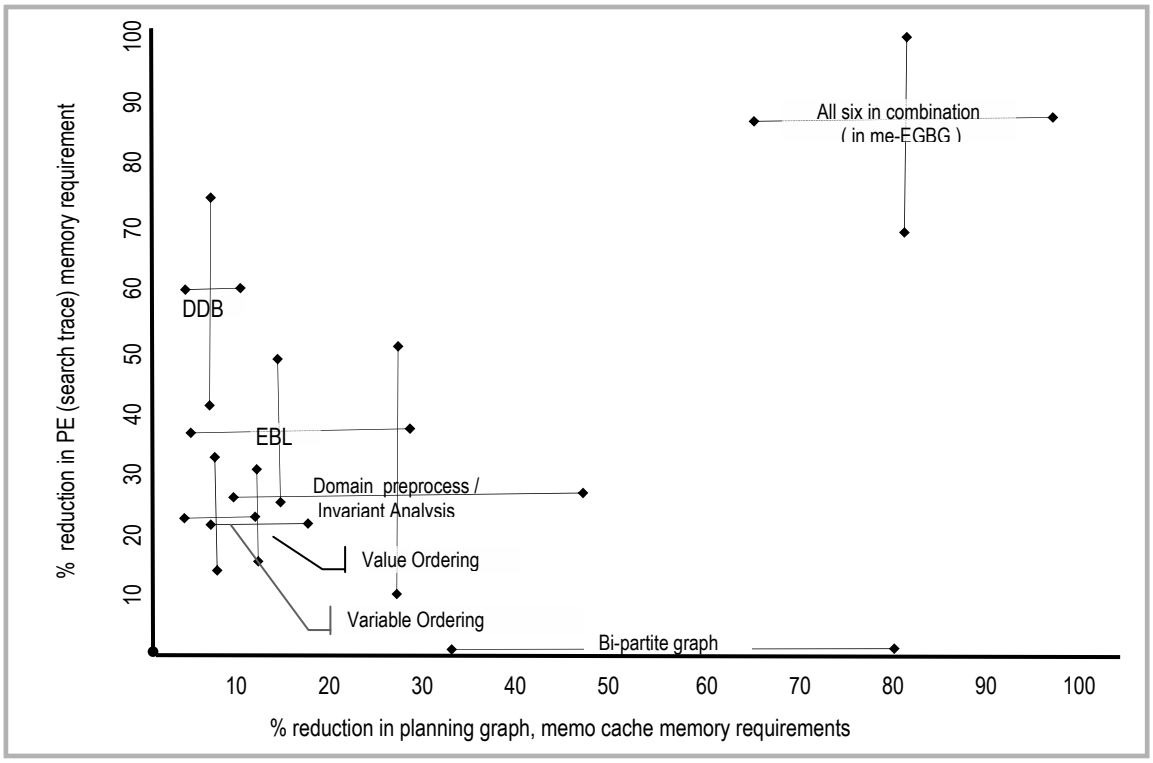

Figure 5: Memory demand impact along two dimensions for six memory reduction/speedup techniques. Plots for each applied independently and as a suite (within EGBG).

set of techniques then, based on their joint average impact on the me-EGBG / PEGG memory footprint over an extensive variety of problems.

Figure 5 illustrates for each method the impact on memory reduction relative to the two dimensions above, *when the method operates in isolation of the others*. The plot reflects results based on twelve problems in three domains (logistics, blocksworld, and tower-of-hanoi), chosen to include a mix of problems entailing large planning graphs, problems requiring extensive search, and problems requiring both. The horizontal axis plots percent reduction in the end-of-run memory footprint of the combined memo caches and the planning graph. The ratios along this ordinate are assessed based on runs with Graphplan (no search trace employed) where the memo cache and planning graph are the only globally defined structures of significant size that remain in the Lisp interpreted environment at run completion.[7] Similarly, the vertical axis plots percent reduction in the space required for the PE at the end of EGBG runs with and without each method activated, and with the planning graph and memo cache structures purged from working memory.

The plot crossbars for each method depict the spread of reduction values seen across the twelve problems along both dimensions, with the intersection being the average. The bi-partite planning graph, not surprisingly, impacts only the graph aspect, but five of the six methods are seen to have an impact on both search trace size and graph/memo cache size. Of these, DDB has the greatest influence on PE size but little impact on the graph or memo cache size, while EBL has a more modest influence on the former and a larger impact on the latter (due both to the smaller memos that it creates

[7] The Allegro Common Lisp 'global scavenging' function was used to purge all but the target global data structures from the workspace.

and the production of more 'general' memos, which engender more backtracks). Domain preprocessing/ invariant analysis can have a major impact on both the graph size and the PE size due to processes such as the extraction of invariants from operator preconditions. It is highly domain dependent, having little effect in the case of blocksworld problems, but can be of great consequence in tower-of-Hanoi and some logistics problems.

That these six methods combined can complement each other is evidenced by the crossbars plotting space reduction when all six are employed at once. Over the twelve problems average reduction in PE size approaches 90% and average reduction in the planning graph/memo cache aspect exceeds 80%. No single method in isolation averages more than a 55% reduction along these dimensions.

The runtime reduction associated with each of these methods in isolation is also highly dependent on the problem and which of the other methods are active. In general, the relative time reduction for any two methods does not correlate closely with their relative memory reduction. However, we found that similarly, the techniques broadly complement each other such that net speedup accrues.

All of the techniques listed above can be (and have been) used to improve Graphplan's performance also, in terms of speed. In order to focus on the impact of planning with the search trace, we use a version of Graphplan that has been enhanced by these six methods for all comparisons to me-EGBG and PEGG in this study (We hereafter refer to this enhanced version of Graphplan as *GP-e)*.

4.2 Experimental Results with me-EGBG

Table 2 illustrates the impact of the six augmentations discussed in the previous section on EGBG's (and Graphplan's) performance, in terms of both space and runtime. Standard Graphplan, GP-e, EGBG, and me-EGBG are compared across 37 benchmark problems in a wide range of domains, including problems from the first three AIPS planning competitions held to date. The problems were selected to satisfy three objectives: a subset that both standard Graphplan and EGBG could solve for comparison to me-EGBG, different subsets that exceed the memory limitations of each of the three planners in terms of either planning graph or PE size, and a subset that gives a rough impression of search time limitations.

Not surprisingly, the memory efficient EGBG clearly outperforms the early version on all problems attempted. More importantly, me-EGBG is able to solve a variety of problems beyond the reach of both standard Graphplan and EGBG. Of the 37 problems, standard Graphplan solves 12, the original EGBG solves 14, GP-e solves 32, and me-EGBG solves 32. Wherever me-EGBG and GP-e solve the same problem, me-EGBG is faster by up to a factor of 62x, and averages ~4x speedup. *Standard* Graphplan (on the twelve problems it can solve), is bested by me-EGBG by factors ranging from 3x to over 1000x.

The striking improvement of the memory efficient version of EGBG over the first version is not simply due to the *speedup* associated with the five techniques discussed in the previous section, but is directly tied to their impact on search trace memory requirements. Table 2 indicates one of three reasons for each instance where a problem is not solved by a planner*:* 1) *s*: planner is still in search after 30 cpu minutes, 2) *pg*: memory is exhausted or exceeded 30 minutes during the planning graph building phase, 3) *pe*: memory is exhausted during search due to pilot explanation extension. The third reason clearly favors me-EGBG as the size of the PE (reported in terms of search segments at the time the problem is solved) indicates that it generates and retains in its trace up to 100x fewer states than EGBG. This translates into a much broader reach for me-EGBG; it exhausts memory on 14% of the

Table 2 problems compared to 49% for the first version of EGBG. Regardless, GP-e solves three problems on which me-EGBG fails in 30 minutes due to search trace memory demands

The table also illustrates the dramatic impact of the speedup techniques on Graphplan itself. The enhanced version, GP-e, is well over 10x faster than the original version on problems they can both solve in 30 minutes, and it can solve many problems entirely beyond standard Graphplan's reach. Nonetheless, me-EGBG modestly outperforms GP-e on the majority of problems that they both can solve. Since the EGBG (and PEGG) planners derive their strength from using the PE to shortcut Graphplan's episodic search process, their advantage is realized only in problems with multiple search episodes and a high fraction of runtime devoted to search. Thus, no speedup is seen for grid-y-1 and all problems in the 'mystery', 'movie', and 'mprime' domains where a solution can be extracted as soon as the planning graph reaches a level containing the problem goals in a non-mutex state.

The bottom-up order in which EGBG visits PE search segments turns out to be surprisingly effective for many problems. For Table 2 problems we found that in the great majority the PE for the final episode contains a seed segment (a state from which search will reach the initial state) within the deepest two or three PE levels. This supports the intuition discussed in Section 3.2 and suggests that the advantage of a low *h-value* bias as observed for heuristic state-space planners (Bonet & Geffner, 1999; Nguyen & Kambhampati, 2000) trans-lates to search on the planning graph.

Results for even the memory efficient version of EGBG reveal two primary weaknesses:

1. The action assignment trace vectors that allow EGBG to avoid redundant search are somewhat costly to generate, make significant demands on available memory for problems that elicit large search (e.g. Table 2 problems: *log-y-4, 8puzzle-1, freecell-2-1*), and are difficult to revise when search experience alters drastically in subsequent visits.
2. Despite its surprising effectiveness in many problems, the bottom up visitation of PE search segments is inefficient in others. For Table 2 problems such as *freecell-2-1* and essentially all 'schedule' domain problems, when the planning graph gets extended to the level from which a solution can be extracted, that solution arises via a *new* search branch generated from the root search segment (i.e. the problem goal state). Thus, the only seed segment in the PE is the top-most search segment, and bottom-up visitation of the PE states is more costly than Graphplan's top-down approach.

The first shortcoming is particularly manifest in problems that do not allow EGBG to exploit the PE (e.g. problems in which a solution can be extracted in the first search episode). The hit that EGBG takes on such problems relative to Graphplan is closely tied to the overhead associated with building its search trace. A compelling tactic to address the second shortcoming is to traverse the search space implicit in the PE according to state space heuristics. We might wish, for example, to exploit any of the variety of state-space heuristics that have revolutionized state space planners in recent years (Bonet & Geffner, 1999; Nguyen & Kambhampati, 2000; Gerevini & Serina, 2002). However, as we noted in Section 3.2, when we depart from a policy of visiting EGBG search segments in level-by-level, bottom-up order, we face more costly bookkeeping and high memory management overhead. More informed traversal of the state-space view of Graphplan's search space is taken up next, where we argue that it's perhaps *the* key benefit afforded by a trace of search on the planning graph.

Problem (steps/actions)	Graphplan cpu sec Stnd.	Graphplan cpu sec GP-e (enhanced)	EGBG cpu sec	EGBG size of PE	me-EGBG (memory efficient EGBG) cpu sec	me-EGBG (memory efficient EGBG) size of PE	SPEEDUP (me-EGBG vs. GP-e)
bw-large-B (18/18)	126	11.4	79	7919	9.2	2090	1.2x
huge-fct (18/18)	165	13.0	98	8410	9.1	2964	1.4x
rocket-ext-a (7/34)	*s*	3.5	40.3	1020	1.8	174	1.9x
att-log-a (11/79)	*s*	12.2	*pe*		7.2	1115	1.7x
gripper-8 (15/23)	125	14.2	88	9790	7.9	2313	1.8x
Tower-6 (63/63)	*s*	43.1	39.1	3303	7.6	80	5.7x
Tower-7 (127/127)	*s*	158	*s*		20.0	166	7.9x
8puzzle-1 (31/31)	667	57.1	*pe*		*pe*	>16000	*(pe)*
8puzzle-2 (30/30)	304	48.3	*pe*		26.9	10392	1.8x
TSP-12 (12/12)	*s*	454	*pe*		97.0	7155	4.7x
AIPS 1998	Graphplan	GP-e	EGBG		me-EGBG		Speedup
grid-y-1 (14/14)	388	16.7	393	19	16.9	15	1x
grid-y-2 (??/??)	*pg*	*pg*	*pg*		*pg*		~
gripper-x-3 (15/23)	291	16.1	200	9888	8.4	2299	1.9x
gripper-x-4 (19/29)	*s*	190	*pe*		65.7	6351	2.9x
gripper-x-5 (23/35)	*s*	*s*	*pe*		433	13572	> 5x
log-y-4 (11/56)	*pg*	470	*pg*		*pe*	>25000	*(pe)*
mprime-x-29 (4/6)	15.7	5.5	6.6	4	5.5	4	1x
movie-x-30 (2/7)	.1	.05	.06	2	.05	2	1x
mysty-x-30 (6/14)	83	13.5	85	32	13.5	19	1x
AIPS 2000	Graphplan	GP-e	EGBG		me-EGBG		Speedup
blocks-10-1 (32/32)	*s*	101	*pe*		16.1	6788	6.3x
blocks-12-0 (34/34)	*s*	24.2	*pe*		14.5	3220	1.7x
logistics-10-0 (15/56)	*s*	30.0	*s*		16.3	1259	1.8x
logistics-11-0 (13/56)	*s*	78.6	*pe*		10.0	1117	7.9x
logistics-12-1 (15/77)	*s*	*s*	*pe*		1205	7101	> 2x
freecell-2-1 (6/10)	*s*	98.0	*pe*		*pe*	>12000	(*pe*)
schedule-8-5 (4/14)	*pg*	63.5	*pg*		42.9	6	1.5x
schedule-9-2 (5/13)	*pg*	58.1	*pg*		46.8	6	1.2x
AIPS 2002	Graphplan	GP-e	EGBG		me-EGBG		Speedup
depot-6512 (10/26)	239	5.1	219	4272	4.1	456	1.25x
depot-7654a (10/28)	*s*	32.5	*s*		14.8	1199	2.2x
driverlog-2-3-6a (10/24)	1280	2.8	807	1569	1.0	230	2.8x
driverlog-2-3-6e (12/28)	*s*	169	*s*		83.3	7691	2x
roverprob1425 (10/32)	*s*	18.9	979	10028	10.3	1522	1.8x
roverprob1423 (9/30)	*s*	170	*pe*		94.7	10217	1.8x
strips-sat-x-5 *(7/22)*	313	47.0	*272*	4111	23.0	2717	2.0x
strips-sat-x-9 *(6/35)*	*s*	*s*	*s*		84.4	306	>21x
ztravel-3-8a (7/25)	*s*	972	*pe*		15.6	1353	62x
ztravel-3-7a (10/21)	*s*	*s*	*pe*		*pe*	>20000	~

Table 2: Search for step-optimal plans: EGBG, me-EGBG, standard & enhanced Graphplan

Standard Graphplan: Lisp version by Smith and Peot

GP-e: Graphplan enhanced per Section 4.1 me-EGBG: memory efficient EGBG

"Size of PE" is the final search trace size in terms of the number of "search segments"

Search failure modes: *'pg'* Exceeded 30 mins. or memory constraints during graph building

'pe' Exceeded memory limit during search due to size of PE

's' Exceeded 30 mins. during search

Parentheses adjacent to cpu time give (# of steps / # of actions) in the solution.

5. Focusing on the State Space View: The so-PEGG and PEGG Planners

The costs associated with EGBG's generation and use of its search trace are directly attributable to the storage, updating, and replay of the CSP value assignments for a search segment's subgoals. We therefore investigated a stripped down version of the search trace that abandons this tactic and focuses instead on the embodied state space information. We will show that the PEGG planners employing this search trace (both so-PEGG, the step-optimal version and PEGG, a version using beam search), outperform the EGBG planners on larger problems. The key difference between EGBG's pilot explanation and the pared down, 'skeletal' PE used by the PEGG planners, is the elimination of the detailed mutex-checking information contained in the bit vectors of the former (i.e. the last item in the bullet list of EGBG search segment contents in Section 3.1). The PEGG planners then apply state-space heuristics to rank the PE search segments based on their associated subgoal sets (states) and are free to visit this 'state space' in a more informed manner. The tradeoff is that for each PE state so visited the planner must regenerate the CSP effort of finding consistent action assignments for the subgoals.

Figure 6 illustrates the PEGG advantage in a small hypothetical search trace at the final search episode. Here search segments in the PE at the onset of the episode appear in solid lines and all plan segments (states extendable to a valid plan) are shown as double-lined boxes. The figure reflects the fact that typically there may be many such latent plan segments in diverse branches of the search trace at the solution-bearing episode. Clearly a planner that can discriminate plan segment states from other states in the PE could solve the problem more quickly than a planner restricted to a bottom-up traversal (deepest PE level first). State space heuristics endow the PEGG planners with this capability.

The so-PEGG planner visits every search segment in the PE during each search episode (comparable to Graphplan's exhaustive search on a given length graph) thereby guaranteeing that returned plans are step-optimal. As such, any advantage of heuristic-guided traversal is realized only in the final episode. For many problems, the computational effort expended by Graphplan in the last search episode greatly exceeds that of all previous episodes combined, so this can still be a powerful advantage. However, as we scale up to problems that are larger in terms of the number and size of search episodes, the cost of exhaustive search in even the intermediate episodes becomes prohibitive. The

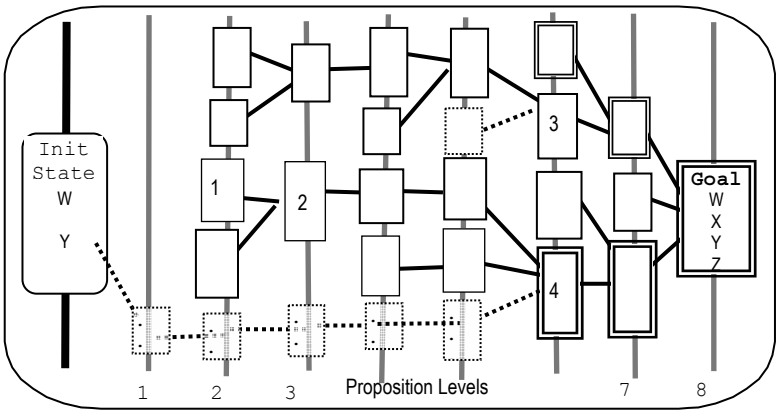

Figure 6: The PE for the final search episode of a hypothetical problem. Search segments in the PE at onset of search appear in solid lines, double-lined boxes represent plan segments, dashed lined boxes are states newly generated in regression search during the episode. Visitation order as dictated by the secondary heuristic is shown via numbering.

planner we refer to simply as PEGG employs beam search, applying the search trace heuristics in all intermediate search episodes to visit only a select subset of the PE segments. In so doing PEGG trades off the step-optimality guarantee for often greatly reduced solution times.

There are several challenges that must be dealt with to effectively use the pared down search trace employed by so-PEGG and PEGG, including adaptation and augmentation of distance-based heuristics to guide search trace traversal and dealing with memory management problems induced by the tactic of 'skipping about the search space'. Before we describe how we addressed such issues and give a more complete description of the algorithm, we first present some results that provide perspective on the effectiveness of these planners.

5.1 Experimental Results With so-PEGG and PEGG

Table 3 compares Graphplan (standard and GP-e), me-EGBG, so-PEGG, and PEGG over most of the same problems as Table 2, and adds a variety of larger problems that only the latter two systems can handle. Table 2 problems that were easily solved for GP-e and me-EGBG (e.g. those in the AIPS-98 'movie' and 'mystery' domains) are omitted from Table 3. Here, all planners that employ variable and value ordering (i.e. all except standard Graphplan), are configured to use value ordering based on the planning graph level at which an action first appears and goal ordering based on proposition distance as determined by the 'adjusted-sum' heuristic (which will be defined below). There are a variety of other parameters for the so-PEGG and PEGG planners for which optimal configurations tend to be problem-dependent. We defer discussion of these to Sections 5.3, 5.4, and 5.6 but note here that for the Table 3 results the following parameter settings were used based on good performance *on average* across a variety of domains and problems:

- Secondary heuristic for visiting states: adjusted-sum with $w_0=1$ (eqn 5-1)
- Beam search: visit the best 20% (lowest f-value) search segments per search episode, with a minimum of 25 and a maximum of 50. Search segments with 'flux' lower than 10% of average are not visited regardless of heuristic rank. (w_{cf} = .01, see section 5.6.1)

Focusing first on the GP-e, me-EGBG, and so-PEGG columns, we clearly see the impact of the tradeoff between storing and exploiting all the intra-segment action assignment information in the PE. In this set of 37 problems, 16 result in me-EGBG exceeding available memory due to the size of the PE while only one pushes that limit for so-PEGG. Seven of the problems that cause me-EGBG to run out of memory are actually solved by so-PEGG while the remainder exceed the time limit during search. In addition, so-PEGG handles five problems in the table that GP-e fails on. These problems typically entail extensive search in the final episode, where the PE efficiently shortcuts the full-graph search conducted by GP-e. The speedup advantage of so-PEGG relative to GP-e ranges between a modest slowdown on three problems to almost 87x on the Zeno-Travel problems, with an average of about 5x. (Note that the speedup values reported in the table are *not* for so-PEGG.)

Generally, any planner using a search trace will under perform GP-e on single search episode problems such as grid-y-1, in which the cost of building the trace is not recovered. The low overhead associated with building so-PEGG's search trace means it suffers little relative to GP-e in this case. On most problems that both me-EGBG and so-PEGG can solve, me-EGBG has the upper hand due to its ability to avoid redundant consistency-checking effort. The fact that me-EGBG's advantage over so-PEGG is not greater for such problems is attributable both to so-PEGG's ability to move about the PE search space in the final search episode (versus me-EGBG's bottom-up traversal) and its lower

Problem	Graphplan		me-EGBG	so-PEGG	PEGG	Speedup
	cpu sec (steps/acts) Stnd.	GP-e (enhanced)	cpu sec (steps/acts)	heur:istic: adjsum cpu sec (steps/acts)	heur: adjsum-u cpu sec (steps/acts)	(PEGG vs. GP-e)
bw-large-B	194.8	11.4 (18/18)	9.2	7.0	4.1 (18/18)	2.8x
bw-large-C	*s*	*s (28/28)*	*pe*	1104	24.2 (28/28)	> 74x
bw-large-D	*s*	*s (38/38)*	*pe*	*pe*	388 (38/38)	> 4.6x
att-log-a	*s*	31.8 (11/79)	7.2	2.9 (11/72)	2.2 (11/62)	14.5x
att-log-b	*s*	*s*	*pe*	*s*	21.6 (13/64)	> 83x
Gripper-8	*s*	14.2 (15/23)	7.9	30.6	5.5 (15/23)	2.6x
Gripper-15	*s*	*s*	*pe*	*s*	46.7 (31/45)	> 38.5x
Tower-7	*s*	158 (127/127)	20.0	14.3	6.1 (127/127)	26x
Tower-9	*s*	*s (511/511)*	232	118	23.6 (511/511)	> 76x
8puzzle-1	2444	57.1 (31/31)	*pe*	31.1	9.2 (31/31)	6.2x
8puzzle-2	1546	48.3 (30/30)	26.9	31.3	7.0 **(32/32)**	6.9x
TSP-12	*s*	454 *(12/12)*	97.0	390	6.9 (12/12)	51x
AIPS 1998	Stnd GP	GP-e	me-EGBG	so-PEGG	PEGG	Speedup
grid-y-1	388	16.7 (14/14)	17.9	16.8	16.8 (14/14)	1x
gripper-x-5	*s*	*s*	433	512	110 (23/35)	> 16x
gripper-x-8	*s*	*s*	*pe*	*s*	520 (35/53)	> 3.5x
log-y-5	*pg*	470 (16/41)	*pe*	361	30.5 (16/34)	15.4x
AIPS 2000	Stnd GP	GP-e	me-EGBG	so-PEGG	PEGG	Speedup
blocks-10-1	*s*	95.4 (32/32)	16.1	18.7	6.9 (32/32)	13.8x
blocks-12-0	~	26.6 (34/34)	14.5	23.0	9.4 (34/34)	2.8x
blocks-16-2	*s*	*s*	*pe*	*s*	28.1 **(56/56)**	> 64 x
logistics-10-0	~	30.0 (15/56)	16.6	21	7.3 (15/53)	4.1x
logistics-12-1	*s*	*s*	1205 (15/77)	1101 (15/75)	17.4 (15/75)	> 103x
logistics-14-0	*s*	*s*	*pe*	*s*	678 (13/74)	> 2.7x
freecell-2-1	*pg*	98.0 (6/10)	*pe*	102	19.5 (6/10)	>92x
freecell-3-5	*pg*	1885 (7/16)	*pe*	511	101 (7/17)	18.7x
schedule-8-9	*pg*	300 (5/12)	615	719	719 (5/12)	(.42x)
AIPS 2002	Stnd GP	GP-e	me-EGBG	so-PEGG	PEGG	Speedup
depot-7654a	*s*	32.5 (10/28)	14.8	12.9	13.2 (10/26)	2.7x
depot-4321	*s*	*s*	*s*	*s*	42.6 (14/37)	>42x
depot-1212	*s*	*s*	*s*	*s*	79.1 (22/53)	>22.8x
driverlog-2-3-6e	*s*	166 (12/28)	83.3	109	80.6 (12/26)	2.1x
driverlog-3-3-6b	*s*	*s*	*pe*	1437 (11/39)	169 **(14/45)**	> 10.7x
roverprob1423	*s*	170 (9/30)	*pe*	63.4	15.0 (9/26)	11.3x
roverprob4135	*s*	*s*	*pe*	s	379 (12 / 43)	> 4.7x
roverprob8271	*s*	*s*	*pe*	s	220 (11 / 39)	> 8.2x
sat-x-5	313	45 (7/22)	43.0	27.0	25.1 (7 / 22)	1.7x
sat-x-9	*s*	*s*	918	9.9	9.9 (6 / 35)	>182x
ztravel-3-8a	*s*	972 (7/25)	15.6	11.2	15.1 **(9/26)**	119x
ztravel-3-7a	*s*	*s*	*pe*	*s*	101 (10/23)	> 18x

Table 3: so-PEGG and PEGG comparison to Graphplan, GP-e, and me-EGBG
GP-e: Graphplan enhanced per Section 4.1 me-EGBG: memory efficient EGBG
so-PEGG: step-optimal, search via the PE, segments ordered by adjusted-sum-u heuristic
PEGG: beam search, best 20% of segments in PE ordered by adjusted-sum-u heuristic
Parentheses give (# of steps/ # of actions) in plan. Boldface values exceed a known step-optimal. See Table 2 for definitions of *s, pg,* and *pe*

overhead due to its more concise search trace. Note that there is no obvious reason to prefer one state traversal order over the other *in non solution-bearing episodes* since these step-optimal planners visit all the states in their PE for these search episodes. [8]

Now turning attention to the PEGG results, it's apparent that the beam search greatly extends the size of problems that can be handled. PEGG solves ten larger problems of Table 3 that could not be solved by either so-PEGG or enhanced Graphplan. Speed-wise PEGG handily outperforms the other planners on every problem except *schedule-8-9*, where GP-e has a factor of 2.3x advantage. As indicated by the table's right-hand column, the speedup of PEGG over GP-e ranges from .42x to over 182x. This is a conservative bound on PEGG's maximum advantage relative to GP-e since speedup values for the seventeen problems that GP-e fails to solve were conservatively assessed at the time limit of 1800 seconds.

We defer further analysis of these results to Section 6 in order to first describe the PEGG algorithm and the advantages it extracts from its search trace.

5.2 The Algorithm for the PEGG Planners

The high-level algorithm for so-PEGG and PEGG is given in Figure 7. As for Graphplan, search begins once the planning graph it has been extended to the level where all problem goals first appear with no binary mutex conditions. (The routine, *find_1st_level_with_goals* is virtually the same as Graphplan's and is not defined here). The first search episode is then conducted in Graphplan fashion, except that the *assign_goals* and *assign_next_level_goals* routines of Figure 8 initialize the PE as they create search segments that hold all states generated during the regression search process. The *assign_goals* pseudo-code outlines the process of compiling "conflict sets" (see Appendix B) as a means of implementing DDB and EBL during the action assignment search. The *assign_next_level_goals* routine illustrates the role of the top-level conflict set for recording a minimal no-good when search on a state is completed (EBL) and depicts how variable ordering need be done only once for a state (when the search segment is created). A child segment is created and linked to its parent (extending the PE) in *assign_next_level_goals* whenever all parent goals are successfully assigned. The *assign_next_level_goals* routine determines the subgoals for the child search segment by regressing the parent's goals over the actions assigned and then checks to see if either the initial state has been reached or there are no remaining goals. If so, success is signaled by returning the child search segment which can then be used to extract the ordered actions in the plan.

Subsequent to the first episode, *PEGG_plan* enters an outer loop that employs the PE to conduct successive search episodes. For each episode, the *newly* generated search segments from the previous episode are evaluated according to a state space heuristic, ranked, and merged into the already ordered PE. In an inner loop each search segment is visited in turn by passing its subgoals to the Graphplan-like *assign_goals* routine.

It is the exit conditions on the inner loop that primarily differentiate so-PEGG and PEGG. Whereas so-PEGG will visit every search segment whose goals are not found to match a memo, PEGG restricts visitation to a best subset, based on a user-specified criterion. As such, expansion of the planning graph can be *deferred* until a segment is chosen for visitation that transposes to a planning graph level exceeding the current graph length. As a consequence, in some problems the PEGG

[8] We have in fact found advantages with respect to traversal order even in intermediate search episodes for some problems. However, this is highly problem-dependent, and we do not consider it in this study.

```
PEGG_PLAN (Ops, Init, Goals)   /*{ Ops, Init, Goals} constitutes a planning problem */
  /* build plangraph, PG, until level n where goals first occur in non-mutex state*/
  Let PG ← find_1st_level_with_goals( Ops, Init, Goals )
  if PG reached level-off and goals are not present in non-mutex state then Return FAIL
  Let n be the number of levels in PG
  Reorder Goals according to variable ordering method
  Let SS0 be a new search segment with fields:
                        goals←Goals,  parent← root,  PE-level← 0,  parent-actions← {}
  Let PE be the pilot explanation structure with fields:  ranked-segs ← {SS0},  new-segs ←{}
  /* Conduct Graphplan-style backward search on the  n-length planning graph, storing trace in PE..*/
  Let search-reslt ← assign_goals(Goals, {}, n, SS0, PG, PE)
  if search-reslt is a search segment   /* Success… */
    then Let Plan ← extract plan actions from the ancestors linked to search-reslt
         Return Plan
   else   /* no n-length solution possible ...use the PE to search for a longer length solution */
     loop forever
      n ← n+1
      /*  rank newly generated states and merge into existing ordered PE segments list */
      PE<ranked-segs> ← merge sort(PE<ranked-segs> ∪ heuristic_sort( PE<new-segs> ) )
      loop while there are unvisited search segments in PE[ranked-segs]  (optionally: for all segments
           below the heuristic threshold)
         Let SS be the highest ranked, unvisited segment from PE's ranked-segs
         Let k = n – (PE-level of SS)      /*... the planning graph level for SS based on transposed PE */
         if k = n then PG ← extend_plangraph(PG)   /*.. delays extending graph until unavoidable! */
           optionally: if flux metric for SS goals < user-specified threshold then continue loop.
         if  SS goals ∈  memos for level k of PG then remove SS from PE's ranked-segs
          else   /* visit search segment SS... */
             search-reslt ← assign_goals (SS<goals>, {}, k, SS, PG, PE)
             if search-reslt is a search segment   /* Success... */
               then Let Plan ← extract plan actions from the ancestors linked to search-reslt
                    Return Plan
              else search-reslt is a conflict set..
                   add conflict set to level k memos of PG   /* memoize the minimal nogood */
                   reorder SS goals so that goals ∈ conflict set appear first   /* EBL-based reordering */
      end while
  end-loop
end
```

Figure 7: Top-level algorithm for PEGG and so-PEGG planners.

planners may be able to extract a step-optimal solution while building one less level than other Graphplan-based planners.[9]

[9] Interestingly, PEGG under beam search could conceivably extract an optimal solution from a planning graph that is an arbitrary number of levels shorter than that required by Graphplan. Consider the case where the PE, on average, extends at least one level deeper in each episode and the subset of PE search segments visited always resides on the deepest levels of the PE. Here an arbitrary number of search episodes might be completed without extending the planning graph. Based on experiments with problems to date however, this advantage seldom saves more than one planning graph level extension.

```
Conduct DDB & EBL-enhanced Graphplan-style search while building search trace
   arguments> G: goals still to be assigned, A: action set already assigned, k: PG level,
              SS1: search segment, PG: planning graph, PE: pilot explanation (search trace)
ASSIGN_GOALS (G, A, k, SS1, PG, PE)
    Let g be a goal selected from G
    Let Ag be the actions from PG level k that support g, ordered by value-ordering heuristic
    Let cs ← {g}   /* initialize a conflict set for DDB */
     loop for act ∈ Ag
        Let search-reslt = {}
        if act is mutex with an action in A
          then Let b be the goal that the conflicted action in A was assigned to support
                cs ← cs ∪ {b}   /* augment the conflict set and continue to loop*/
         else   /* act has no conflict with actions already in A */
            if G-{g} is not empty
               then   /* continue goal assignment at this level… */
                    search-reslt ← assign_goals (G-{g}, A ∪{act}, k, SS1, PG, PE)
             else   /* there are no SS1 goals left to satisfy...setup for search at the next lower level */
                  search-reslt ← assign_next_level-goals (A ∪{act}, k, SS1, PG, PE)
            if search-reslt is a conflict set, check if it contains current goal..
               if g ∈ search-reslt
                 then   /* absorb returned conflict set & try next action */
                      cs ← cs ∪ search-reslt
                 else Return search-reslt    /*just return this conflict set */
              else search-reslt is a search segment:   /* Success.. */
                   Return search-reslt
     end loop (actions)
    Return cs    /* no soln reached .. compiled conflict set is returned*/
 end-if
end

Set up search on graph level k-1 given that SS1 goals have been satisfied by actions in A at level k
ASSIGN_NEXT_LEVEL_GOALS (A, k, SS1, PG, PE)
 Let nextgoals ← regress SS1 goals over A (the actions assigned to satisfy goals)
 if nextgoals is empty or k = 0 (it's the initial state)
    then Return SS1   /* Success */
 else if there is an M ∈ memos at level k-1 of PG such that M ⊆ nextgoals
      then Return M as the conflict set    /* backtrack due to nogood */
 else   /* … initiate search on the next lower PG level*/
    Let SS2 be a new search segment holding nextgoals, a pointer to SS1, & actions A assigned in SS1
     Add SS2 to PE new-segs list
     Let search-reslt ← assign_goals (nextgoals, {}, k-1, SS2, PG, PE)
     if search-reslt is a search segment   /* Success.. */
        then Return search-reslt
       else search-reslt is a conflict set:   /*memoize minimal nogood and return conflict set */
            add conflict set to level k-1 memos of PG
            reorder SS2 goals such that goals ∈ conflict set appear first   /* .. EBL-based reordering */
            Return search-reslt
 end-if
end
```

Figure 8: PEGG / so-PEGG regression search algorithm for Graphplan-style regression search on subgoals while concurrently building the search trace (PE)

Note that PEGG's algorithm combines both state-space and CSP-based aspects in its search:

- It chooses for expansion the most promising state based on the previous search iteration and state space heuristics. PEGG and so-PEGG are free to traverse the states in its search trace in any order.
- A selected state is expanded in Graphplan's CSP-style, depth-first fashion, making full use of all CSP speedup techniques outlined above.

The first aspect most clearly distinguishes PEGG from EGBG: traversal of the state space in the PE is no longer constrained to be bottom-up and level-by-level. As it was for EGBG, management of memory associated with the search trace is a challenge for PEGG once we stray from bottom-up traversal, but it is less daunting. It will be easier to outline how we address this if we first discuss the development and adaptation of heuristics to search trace traversal.

5.3 Informed Traversal of the Search Trace Space

The HSP and HSP-R state space planners (Bonet & Geffner, 1999) introduced the idea of using the 'reachability' of propositions and sets of propositions (states) to assess the difficulty degree of a relaxed version of a problem. This concept underlies their powerful 'distance based' heuristics for selecting the most promising state to visit. Subsequent work demonstrated how the planning graph can function as a rich source of such heuristics (Nguyen & Kambhampati, 2000). Since the planning graph is already available to PEGG, we adapt and extend heuristics from the latter work to serve in a *secondary heuristic* role to direct PEGG's traversal of its search trace states. Again, the primary heuristic is the planning graph length that is iteratively deepened (Section 2.2), so the step-optimality guarantee for the so-PEGG planner does not depend on the admissibility of this secondary heuristic.

There are important differences between heuristic ranking of states generated by a state space planner and ordering of the search segments (states) in PEGG's search trace. For example, a state space planner chooses to visit a given state only once while the PEGG planners often must consider whether to *revisit* a state in many consecutive search episodes. Ideally, a heuristic to rank states in the search trace should reflect level-by-level evolutions of the planning graph, since the transposition process associates a search segment with a higher level in each successive episode. For each higher planning graph level that a given state is associated with, the effective regression search space 'below' it changes as a complex function of the number of new actions that appear in the graph, the number of dynamic mutexes that relax, and the no-goods in the memo caches. Moreover, unlike a state space planner's queue of previously *unvisited* states, the states in a search trace include all children of each state generated when it was last visited. Ideally the value of visiting a state should be assessed independently of the value associated with any of its children, since they will be assessed in turn. Referring back to the search trace depicted in Figure 6, we desire a heuristic that can, for example, discriminate between the #4 ranked search segment and its ancestor, top goal segment (WXYZ). Here we would like the heuristic assessment of segment WXYZ to *discount* the value associated with its children already present in the trace, so that it is ranked based only on its potential for generating new local search branches.

We next discuss adaptation of known planning graph based heuristics for the most effective use with the search trace.

5.3.1 ADOPTION OF DISTANCE-BASED STATE SPACE HEURISTICS

The heuristic value for a state, *S,* generated in backward search from the problem goals can be expressed as:

5-1) $f(S) = g(S) + w_0 * h(S)$

where: *g(S)* is the distance from *S* to the problem goals (e.g. in terms of steps)
h(S) is a distance estimate from *S* to the initial state (e.g. in steps)
w_0 is an optional weighting factor

The value of *g* for any state generated during the search (e.g. the states in the PE) is easily assessed as the cumulative cost of the assigned actions up to that point. The *h* values we consider here are taken from the distance heuristics adapted to exploit the planning graph by (Nguyen & Kambhampati, 2000). One heuristic that is readily extractable from the planning graph is based on the notion of the level of a set of propositions:

Set Level heuristic: *Given a set S of propositions, denote lev(S) as the index of the first level in the leveled serial planning graph in which all propositions in S appear and are non-mutex with one another. (If S is a singleton, then lev(S) is just the index of the first level where the singleton element occurs.) If no such level exists, then lev(S) =* ∞.

This admissible heuristic embodies a lower bound on the number of actions needed to achieve *S* from the initial state and also captures some of the negative interactions between actions (due to the planning graph binary mutexes). In the Nguyen & Kambhampati, 2000 study, the set level heuristic was found to be moderately effective for the backward state space (BSS) planner AltAlt, but tended to result in too many states having the same f-value. In directing search on PEGG's search trace it is somewhat more effective, but still suffers from a lower level of discrimination than some of the other heuristics they examined -especially for problems that engender a planning graph with relatively few levels. Nonetheless, as noted in the Appendix B discussion of memory efficiency improvements we use it during planning graph construction as the default heuristic for value ordering, due to both its low computational cost and its synergy with building and using a bi-partite planning graph.

The inadmissible heuristics investigated in the Nguyen & Kambhampati, 2000 work are based on computing the heuristic cost *h(p)* of a single proposition iteratively to fixed point as follows. Each proposition *p* is assigned cost 0 if it is in the initial state and ∞ otherwise. For each action, *a*, that adds *p*, *h(p)* is updated as:

5-2) $h(p) := \min\{ h(p), 1+h(Prec(a)) \}$

where *h(Prec(a))* is the sum of the h values for the preconditions of action *a*.

Given this estimate for a proposition's h-value, a variety of heuristic estimates for a state have been studied, including summing the *h* values for each subgoal and taking the maximum of the subgoal h-values. For this study we will focus on a heuristic termed the 'adjusted-sum' (Nguyen & Kambhampati, 2000), that combines the set-level heuristic measure with the sum of the h-values for a state's goals. Though not the most powerful heuristic tested by them, it is computationally cheap for a planning graph based planner and was found to be quite effective for the BSS planners they tested.

Adjusted-sum heuristic*: Define lev(p) as the first level at which p appears in the plan graph and lev(S) as the first level in the plan graph in which all propositions in state S appear and are non-mutexed with one another. The adjusted-sum heuristic may be stated* as:

5-3) $$h_{adjsum}(S) := \sum_{p_i \in S} h(p_i) + (lev(S) - \max_{p_i \in S} lev(p_i))$$

This is a 2-part heuristic; a summation, which is an estimate of the cost of achieving S under the assumption that its goals are independent, and an estimate of the cost incurred by negative interactions amongst the actions that must be assigned to achieve the goals. The latter factor is estimated by taking the difference between the planning graph level at which the propositions in S first become non-mutex with each other and the level in which these propositions first appear together in the graph.

More complex heuristics have been proposed that include a measure of the positive interactions between subgoals in a state, that is, the extent to which an action establishes more than one relevant subgoal. The so-called 'relaxed plan' distance-based heuristics focus on the positive interactions, and several studies have demonstrated their power for backward and forward state-space planners (Nguyen & Kambhampati, 2000; Hoffman, 2001). However, as reported in the former study, the primary advantage of adding positive interactions to the adjusted-sum heuristic is to produce shorter make-span plans at the expense of a modest increase in planning time. Since PEGG's IDA* search already ensures optimal make-span there is little incentive to incur the expense of the relaxed plan calculation, and we restricted our work here to the simpler adjusted-sum heuristic of eqn 5-3.

The adjusted-sum heuristic can be further adapted to search on the planning graph by leveraging the information in PEGG's search trace. This takes the form of heuristic updating to dynamically improve the h value estimate of states in the PE. The *lev(S)* term in the adjusted-sum heuristic represents the first planning graph level at which the subgoals in state *S* appear and are *binary* non-mutex with each other. However, once regression search on *S* at graph level *k* fails in a given episode, the search process has essentially discovered an n-ary mutex condition between some subset of the goals in *S* at level *k* (This subset is the conflict set, C, that gets memoized in the PEGG algorithm of Figures 7 and 8). At this point the *lev(S)* value can be updated to *k+1,* indicating that *k+1* is a conservative estimate of the first level that the *S* goals appear in n-ary non-mutex state. This has a desirable property for ranking search trace states; the longer a state resides in the search trace, the more often its h-value gets increased, and the less appealing it becomes as a candidate to visit again. That is, this heuristic update biases against states that have been visited the most and failed to extend to a solution. We use this augmented adjusted-sum heuristic for the PEGG runs in this work and refer to it as "adjusted-sum-u".

Experimentally, we find that the advantage of any given heuristic for ordering PE states is highly domain dependent (but less sensitive to a particular domain problem). For example, compared to a simple bottom-up visitation strategy, the adjusted-sum-u heuristic improves so-PEGG runtimes by up to an order of magnitude in some domains (e.g. *Freecell* and*, Satellite*) while degrading it by up to a factor of 2x to 7x in others (e.g. *Zenotravel*). Figure 9 depicts the performance of the adjusted-sum-u heuristic relative to a bottom-up heuristic in so-PEGG on several sets of problems. Here the heuristics are compared in terms of so-PEGG's average computation time as a percentage of GP-e's in the final search episode -the most important measure for exhaustive search on the planning graph. The more informed heuristic will not only find a seed segment sooner but, in the event there are many (typical of logistics domains), it will find one that lies on a planning graph level that is closer to the initial state. A less informed heuristic may cause PEGG to end up conducting more search in its final episode than GP-e, as there may be many states in the PE that would not be regenerated by Graphplan in its final regression search before it finds a solution. This is a direct measure of the power of the

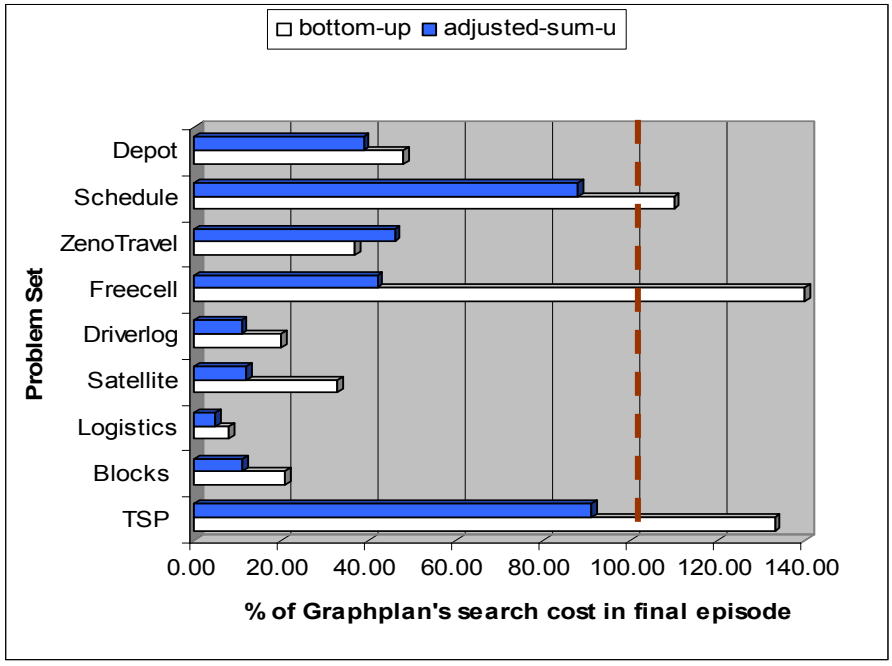

Figure 9: Heuristic accuracy in so-PEGG for the final search episode --relative to GP-e

search segment selection heuristic. Since performance can vary considerably with the specific problem the results in the figure are averages for three representative examples in each domain[10].

5.4 Memory Management Under Arbitrary Search Trace Traversal Order

We return now to the memory management problems induced by the strategy of skipping about the search space. Consider again the PE at the time of the final search episode in Figure 6. If search segments are visited in an order other than deepest PE level first, we encounter the problem of regenerating states that are already contained in the PE. The visitation order depicted by numbered segments in the figure could result from a fairly informed heuristic (the 4th segment it chooses to visit is a plan segment), but it implies that many states already resident in the PE will be regenerated. This includes, for example, all the as yet unvisited descendents of the third segment visited. Unchecked, this process can significantly inflate search trace memory demands as well as the overhead associated with regenerating search segments. In addition, heuristic information about a state is lost when the state is regenerated instead of revisited as an extant PE search segment. This is because due to the adjusted-sum-u secondary heuristic PEGG 'learns' an improved n-ary mutex level for a search segment's goals and updates its f-value accordingly in each search episode.

We address this issue by hashing every search segment generated into an associated PE state hash table according to its canonically ordered goal set. One such hash table is built for each PE level. Prior to initiating regression search on a subgoal set of a search segment, S_n , PEGG first checks the planning graph memo caches and, if no relevant memo is found, it then checks the PE state hash table to see if S_n's goals are already embodied in an existing PE search segment at the relevant PE level. If

[10] Problem sets used- Blocksworld: *bw-large-b, blocks-10-1 and 12-0* Logistics: *att-log-a, logistics-10-0, 12-0,* Gripper: *gripper8, gripper-x-3, x-5,* Depot: *depotprob6512, -5646, 6587*, Driverlog: *dlog-3-36a, -2-3-6a, 2-3-6e*, Zenotravel: *ztravel-3-8a, -3-8b, 3-7b*, Freecell: *freecell-2-1, -2-2, -3-5,* Satellite: *strips-sat-x-4, x-5, x-9.*

such a search segment, S_e, is returned by this PE state check, S_e is made a child of S_n (if it is not already) by establishing a link, and search can then proceed on the S_e goals.[11]

Another search trace memory management issue is associated with the fact that PEGG only visits a subset of the PE states -a set we will call the "active" PE. It is tempting to pursue a minimal memory footprint strategy by retaining in memory only the active search segments in the PE. However unlike Graphplan, when the initial state is reached PEGG cannot extract a solution by unwinding the complete sequence of action assignment calls since it may have begun this regression search from an arbitrary state in any branch of the search trace tree. PEGG depends instead on the link between a child search segment and its parent to extract the plan actions once a solution is found. As such we must retain as a minimum, the active search segments and all of their ancestor segments up to the root node.

Beyond the requirement to retain search segments tied to the active PE, there are many strategies that might be used in managing the inactive portion. For this study we have not attempted to reduce PE memory requirements in this manner, instead focusing on what might be termed the search space 'field of view'. Under beam search, heuristic effectiveness depends not only on how informed it is, but the search trace states available for it to rank. The reduced memory footprint of PEGG's skeletal search trace allows us to adopt a strategy of retaining in memory *all* search segments generated. All such segments have their f-values updated before they are ranked, giving the beam search a wide selection of states contending for 'active' status in a given search episode.

5.5 Learning to Order a State's Subgoals

The PEGG planners employ *both* EBL and a search trace, and this allows them to overlay a yet more sophisticated version of variable ordering on top of the distance-based ordering heuristic. The guiding principle of variable ordering in search is to fail early, when failure is inevitable. In terms of Graphplan-style search on a regressed state, this translates as 'Since all state goals must be assigned an action, it's best to attempt to satisfy the most difficult goals first.' The adjusted-sum heuristic described above, applied to a single goal, provides an estimate of this difficulty based on the structure of the planning graph. However, EBL provides additional information on the difficulty of goal achievement based directly on search experience. To wit, the conflict set that is returned by the PEGG's *assign_goals* routine during search on a goal set explicitly identifies which of these goals were responsible for the search failure. The intuition behind the EBL-based reordering technique then, is that these same goals are likely to be the most difficult to assign when the search segment is revisited in the next search episode. This constitutes a dynamic form of variable ordering in that, unlike the distance-based ordering, a search segment's goals may be reordered over successive search episodes based on the most recent search experience.

Figure 10 compares the influence of adjusted sum variable ordering and EBL-based reordering methods on memory demand, in a manner similar to Figure 5. Here the impact of EBL-based reordering on EGBG's performance is reported because PEGG tightly integrates the various CSP and efficiency methods, and their independent influence cannot be readily assessed.[12] We isolate the impact of EBL-based reordering from that of EBL itself by activating the EBL but using the produced con-

[11] In the interests of simplicity, the Figure 8 algorithm does not outline this memory management process.

[12] Given the success of the various memory-efficiency methods within EGBG, all versions of PEGG implement them by default. A graph analogous to Figure 5 for the PEGG planner would differ in terms of the actual memory reduction values, but we are confident that the overall benefits of the methods would persist, as would the relative benefit relationship between methods.

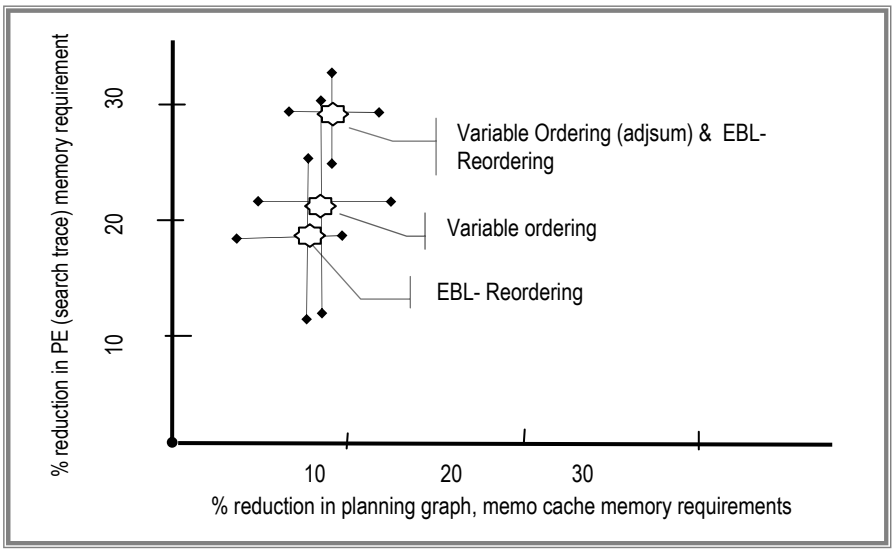

Figure 10: Memory demand impact along two dimensions for 'adjusted-sum' variable ordering and EBL-based reordering techniques when applied independently and together.

flict sets only in reordering, not during memoization. The average reduction in search trace memory over the 12-problem sample is seen to be about 18% for EBL-based reordering alone. This compares favorably with the 22% average reduction of the distance-based ordering, especially since, unlike the adjusted sum ordering, the EBL-based reordering *only takes effect in the 2nd search episode*. The plot also reveals that the two modes of ordering are quite complimentary.

Across a variety of problems and domains we found the following approach to be most effective in combining distance-based variable ordering and EBL-based reordering: 1) a newly created search segment's goals are ordered according to the distance-based heuristic. 2) After each visit of the search segment, the subset of its goals that appear in the conflict set are reordered to appear first. 3) The goals in the conflict set are then ordered by the distance-based heuristic and appended to non-conflict goals, which are also set in distance-based order.

As indicated in Figure 10, this hybrid form of variable ordering not only boosts the average memory reduction to almost 30%, but also significantly reduces the wide fluctuation in performance of either method in isolation. We re-emphasize here that this search experience-informed goal ordering is only available to a search algorithm that maintains a memory of states it has visited. It is therefore not portable to any Graphplan-based planner we know of.

5.6 Trading Off Guaranteed Step-Optimality for Speed and Reach: PEGG Under Beam Search

Many of the more difficult benchmark problems for Graphplan's IDA* style search have 20 or more such search episodes before the reaching the episode in which a solution can be extracted. The cumulative search time tied to these episodes can be a large portion of the total search time and, as indicated in Table 3, so-PEGG exhausts search time limits well before reaching the episode in which a solution can be extracted. The strategy of exhaustively searching the planning graph, in each episode up to the solution bearing level, gives the step-optimal guarantee to Graphplan's solutions but it can exact a

high cost to ensure what is, after all, only one aspect of plan quality. We explore with PEGG, a non-exhaustive search version of so-PEGG, the extent to which search episodes can be truncated while producing plans with virtually the same makespan as Graphplan's solution.

PEGG shortcuts the time spent in search during these intermediate episodes by using the secondary heuristic to not only direct the order in which PE states are visited but to prune the search space visited in the episode. This beam search seeks to visit only the most promising PE states, as measured by their f-values against a user-specified limit. In addition, the beam search has an important dual benefit for PEGG in that it further reduces the memory demands of its search trace and, depending on the problem, even the planning graph. In the PEGG algorithm of Figure 8, the 'loop while' statement is the point at which a beam search f-value threshold can be optionally applied to PE states that are candidates for visitation. When the first segment exceeding this threshold is reached on the sorted queue the search episode ends.

To devise an effective threshold test must reconcile competing goals: minimizing search in non-solution bearing episodes while maximizing the likelihood that the PE retains and visits (preferably as early as possible), a search segment that's extendable to a solution once the graph reaches the *first level* with an extant solution. The narrower the window of states to be visited, the more difficult it is for the heuristic that ranks states to ensure it includes a plan segment, i.e. one that is part of a step-optimal plan. PEGG will return a step-optimal plan as long as its search strategy leads it to visit *any* plan segment (including the top, 'root' segment in the PE) belonging to *any* plan latent in the PE, during search on the first solution-bearing planning graph. The heuristic's job in selecting the window of search segments to visit is made *less* daunting for many problems because there are *many* step-optimal plans latent at the solution-bearing level.

We next describe an effective planning graph based metric that augments the state space heuristic in choosing the set of PE states to visit in each search episode.

5.6.1 MINING THE PLANNING GRAPH TO FILTER THE BEAM

Beyond the heuristic updating we introduced in Section 3, the distance-based heuristics are virtually insensitive to planning graph evolution as a search segment is transposed up successive levels. Since the search trace contains all children states that were generated in regression search on a state S in episode n, a heuristic preference to include S over other states in the trace to visit in episode $n+1$ should reflect the chance that it will directly generate *new* and promising search branches. The child states of S from search episode n are *competitors* with S, so ideally a heuristic's rank for S should reflect in some sense the value of visiting the state *beyond* the importance of its children.

Consider now the sensitivity of the adjusted-sum heuristic (or any of the distance-based heuristics) to possible differences in the implicit regression search space 'below' a set of propositions, S, at planning graph level k versus level $k+1$. Given that the propositions are present and binary non-mutex with each other at level k, only the cost summation factor in equation 5-3 could conceivably change when S is evaluated at level $k+1$. This would require two conditions: a new action must establish one of the S propositions for the first time at level k+1 and the action's precondition costs must sum to *less* than the precondition costs of any other establisher of the proposition. In practice this happens infrequently since the later that an action appears in the graph construction process, the higher its cost tends to be. Consequently h-values for states based on distance-based heuristics remain remarkably constant for planning graph levels beyond that at which the propositions appear and are binary non-

mutex[13]. We desire a means of compensating for a static h-value when a state is transposed to a planning graph level at which promising new branches of regression search open up.

The likelihood that a state visited in episode *n* at graph level *k* will give rise to new child states if visited in episode *n+1* at level *k+1* is rooted in the graph dynamics summarized by Observations A-1 and A-2 of Appendix A. Three planning graph and memo cache properties determine whether regression search on a subgoal set *S* will evolve over successive episodes:

1. There are new actions at level *k+1* that establish a subgoal of *S*
2. There are dynamic mutexes at level *k* between actions establishing subgoals of *S* that relax at level *k+1*
3. There were no-good memos encountered in regression search on state *S* during episode *n* that will not be encountered at level *k+1* (and also the converse).

This set of measures of the potential for new search branches to result from visiting a state in the PE we refer to as the 'flux'; the intuition being that the higher the flux, the more likely that search on a given state will differ from that seen in the previous search episode (and captured in the PE). If none of the three factors applies to a state under consideration, there is no point in visiting it, as no new search can result relative to the previous episode.

The first factor above can be readily assessed for a state (thanks in part to the bi-partite graph structure). The second flux factor is unfortunately expensive to assess; a direct measure requires storing all pairs of attempted action assignments for the goals of *S* that were inconsistent in episode *n* and retesting them at the new planning graph level. However, the graph mechanics are such that relaxation of a dynamic mutex between two actions at level *k requires* the relaxation of a dynamic mutex condition between some pair of their preconditions at level *k-1* (one precondition for each action). This relaxation, in turn, is either due to one or more new establishing actions for the preconditions at level *k-1* or recursively, relaxations in existing actions establishing the preconditions. As such, the number of new actions establishing the subgoals of a state *S* in the PE (factor 1 above) provide not only a measure of the flux for *S,* but also a predictor of the flux due to factor 2 for the parent (and higher ancestors) of *S.* Thus, it turns out that by simply tracking the number of new actions for each state subgoal at its current level and propagating an appropriately weighted measure up to its parent, we can compile a useful estimate of flux for factors 1 and 2 above.

The third flux factor above is the most unwieldy and costly to estimate; an exact measure requires storing all child states of *S* generated in regression search at level *k* that caused backtracking due to cached memos, and retesting them to see if the same memos are present at level *k+1*.[14] Ignoring this factor, we sum just the two flux measures that depend on new actions to derive a filtering metric which is then used to assist the largely static adjusted-sum distance-based heuristic in culling the beam. The resulting (inexact) metric is sensitive to the evolution in search potential as a state is transposed to higher planning graph levels:

[13] This, in part, explains the observation (Nguyen & Kambhampati, 2000) that the AltAlt state space planner performance generally degrades very little if the planning graph used to extract heuristic values is built only to the level where the problem goals appear and are non-mutex, rather than extending to level-off.

[14] Note that as long as we are using EBL/DDB, it is not sufficient to just test whether *some* memo exists for each child state. This is because no-good goals themselves contribute to the conflict set used to direct search within *S* whenever such backtracking occurs.

5-4) $$flux(S) = \frac{\sum_{p_i \subset S} newacts(p_i)}{|S|} + w_{cf} * \sum_{s_i \subset S_c} childflux(s_i)$$

where: p_i is a proposition in state S

$newacts(p_i)$ is the number of new actions that establish proposition p_i of S at its associated planning graph level

$|S|$ is a normalization factor; the number of propositions in S

S_c is the set of all child states of S currently represented in the search trace

$childflux(s_i)$ is the sum of the two flux terms in eqn 5-4 applied to child state s_i of S

w_{cf} weights the contribution of flux from child states to the parent state

Here the number of new actions establishing the subgoals of a state is normalized relative to the number of subgoals in the state.

We report elsewhere (Zimmerman, 2003) on the use of this flux to directly augment the secondary heuristic. Depending on the domain and the weighting of flux contribution to the adjusted-sum heuristic, speedups of up to an order of magnitude are observed[15]. However its impact is highly domain dependent and since we are primarily concerned with the performance of a general purpose planner, in this study we only consider its use as a beam filter.

Under beam search, the flux measure can strongly impact every search episode, as it influences the states actually included in the active PE. When used in this mode, search segments with an assessed flux below a specified threshold are skipped over even if their f-value places them in the active PE. Flux proves to be broadly effective across most domains when used in this mode. As mentioned in Section 5.1, we use a flux cutoff in each search episode of 10% of the average flux for all search segments in the PE; any segment below this value is not visited regardless of its heuristic rank. At this setting the impact on speedup for the PEGG column problems of Table 3 ranges from nil to a factor of 9x over PEGG's performance without the flux filter. Higher settings can dramatically speed up the solution search, but often at the expense of greater solution makespan.

5.6.2 PEGG'S ABILITY TO FIND STEP-OPTIMAL PLANS

The variety of parameters associated with the beam search approach described above admits considerable flexibility in biasing PEGG towards producing plans of different quality. Shorter makespan plans are favored by more extensive search of the PE states in each episode while heuristically truncated search tends to generate non-optimal plans more quickly, often containing redundant or unnecessary actions. The settings used in this study clearly bias PEGG solutions towards step-optimality: The step-optimal plan produced by enhanced Graphplan is matched by PEGG for all but four of the 37 problems reported in Table 3, as indicated by the annotated steps and actions numbers given in parenthesis next to successful GP-e and PEGG runs[16]. (PEGG solutions with a longer makespan than the step-optimal have boldface step/action values.) In these four problems, PEGG returns solutions

[15] For example, compared to a simple bottom-up visitation strategy, the flux-augmented adjusted-sum heuristic improves so-PEGG runtimes by up to 11x in some domains (e.g. *Freecell* and, *Satellite*) while degrading it by as much as 2x to 7x in others (e.g. *Zenotravel*).

[16] Where more than one of the guaranteed step-optimal planners (GP-e, me-EGBG and so-PEGG) finds a solution the steps and actions are reported only for one of them, since they all will have the same makespan.

Problem	'N-best first' [N=100, state-space search]	PEGG [adjusted-sum-u heuristic with beam search on the 100 best search segments]	SATPLAN (optimal)
bw-large-a	8	6 (.1 s)	6
bw-large-b	12	9 (4.5 s)	9
bw-large-c	21	14 (39.0 s)	14
bw-large-d	25	18 (412 s)	18

Table 4: Quality comparison (in terms of plan steps) for PEGG and N-best beam search on a forward state space planner (Bonet et al., 1997).

within four steps of optimum, in spite of its highly pruned search. This proved to be a fairly robust property of PEGG's beam search under these settings across all problems tested to date.

PEGG under the adjusted-sum-u secondary heuristic often finds plans with fewer *actions* than GP-e in parallel domains, and this 'Graphplan hybrid' system is also impressive in serial domains such as blocksworld (which are not exactly Graphplan's forte).

The tactic of trading off optimal plan length in favor of reduced search effort is well known in the planning community. By comparison, PEGG's beam search approach is biased towards producing a very high quality plan at possibly some expense in runtime. For example, in their paper focusing on an action selection mechanism for planning, Bonet et. al. briefly describe some work with an "N-best first algorithm" (Bonet, Loerincs, & Geffner, 1997). Here they employ their distance-based heuristic to conduct beam search in forward state space planning. They report a small set of results for the case where the 100 best states are retained in the queue to be considered during search.

Table 4 reproduces those results alongside PEGG's performance on the same problems using beam search. Here, to approximate the N-best algorithm, PEGG is also run with 100 states visited in each intermediate search episode. The 1997 study compared the N-best first approach against SATPLAN, which produces the optimal length plan, to make the point that the approach could produce plans reasonably close to optimal with much less search. The 'N-best first' code is not available to run on our test platform, so only PEGG's runtime is reported. Focusing on plan makespan, it's clear that even in this serial domain, the parallel planner PEGG produces a much shorter plan than the 'N-best first' state space approach, and in fact finds the optimum length plan generated by SATPLAN in all cases.

More recently LPG (Gerevini & Serina, 2002), another planner whose search is tightly integrated with the planning graph, was awarded top honors at the AIPS-2002 planning competition, due to its ability to quickly produce high quality plans across a variety of domains currently of interest. In the Figure 11 scatter plot, solution quality in terms of steps for LPG and PEGG are compared against the optimal for 22 problems from three domains of the 2002 AIPS planning competition. We chose these particular problems because the optimal solution is known, and we are interested in comparing to a quality baseline. LPG's results are particularly apt in this case, because that planner also non-exhaustively searches the planning graph at each level before extending it, although its search process differs markedly from PEGG's. LPG, too, can be biased to produce plans of higher quality (generally at the expense of speed) and here we report the competition results for its 'quality' mode. In terms of number of actions in the solutions neither planner consistently dominates for these problems but PEGG clearly excels in step-optimality. Its maximum deviation from optimum is four steps and most

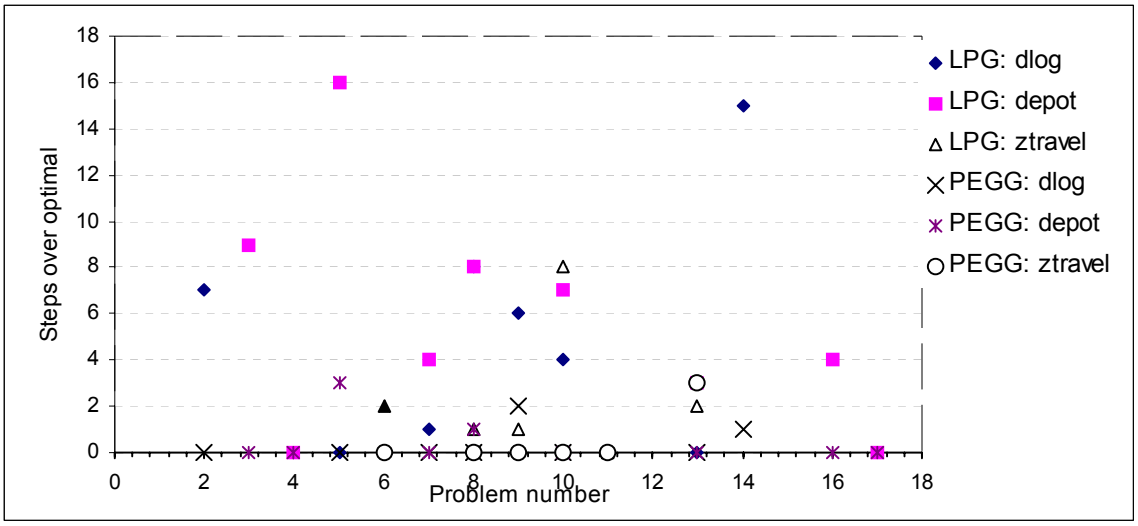

Figure 11: Makespan comparison of PEGG and LPG -Departure from step-optimal plan length. (LPG data taken from the AIPS 2002 competition results.)

of the plot points for its solutions lie right on the optimal makespan axis. It is possible that there are sets of actions within LPG's solutions that could be conducted in parallel but the algorithm's quality mode heuristic is insensitive to them .

It should be noted that LPG produced solutions for some difficult problems in these domains that PEGG currently does not solve within a reasonable time limit. We are investigating the characteristics of these problems that make them so difficult for PEGG.

5.6.3 PEGG COMPARED TO HEURISTIC STATE SPACE SEARCH

We have not attempted here to run PEGG head-to-head for speed against recent IPC planners, in part due to platform difficulties (PEGG is written in Lisp while the competition planners are generally coded in C and published results are based on the execution on the competition machines) and partly due to our focus on near-optimal makespan for parallel plans rather than speed. Given PEGG's close coupling with the planning graph, the most relevant comparisons are with other parallel planners that also employ the graph in some form. For such comparisons, we would like to isolate the search component of the runtime from planning graph construction, since there are a variety of routines that produce essentially the same graph with widely different expenditures of computational time and memory. The reported runtimes for the LPG planner in the AIPS-02 competition are generally much smaller than PEGG's, but it's difficult to isolate the impact of graph construction and platform-related effects, not to mention the disparity in the makespan of the plans produced.

Table 5 compares PEGG against a Lisp version of a fast distance-based heuristic state space planner using most of the same problems as Table 3. AltAlt (Srivastava et al., 2001), like PEGG, depends on the planning graph to derive the powerful heuristics it uses to direct its regression search on the problem goals. This facilitates planner performance comparison based on differences in search without confusing graph construction time issues. The last column of Table 5 reports AltAlt performance (runtime and makespan) for two of the most effective heuristics developed for that planner (Nguyen & Kambhampati, 2000), the first of which is the adjusted-sum heuristic as described in Section 5.3.1.

Problem	PEGG heuristic: adjusted-sum-u cpu sec (steps/acts)	Alt Alt (Lisp version) cpu sec (/ acts) heuristics: adjusum2	combo
bw-large-B	4.1 (18/18)	67.1 (/ 18)	19.5 (/28)
bw-large-C	24.2 (28/28)	608 (/ 28)	100.9 (/38)
bw-large-D	388 (38/38)	950 (/ 38)	~
rocket-ext-a	1.1 (7/34)	23.6 (/ 40)	1.26 (/ 34)
att-log-a	2.2 (11/62)	16.7 (/56)	2.27(/ 64)
att-log-b	21.6 (13/64)	189 (/ 72)	85.0 (/77)
Gripper-8	5.5 (15/23)	6.6 (/ 23)	*
Gripper-15	46.7 (36/45)	10.1 (/ 45)	6.98 (/45)
Gripper-20	1110.8 (40/59)	38.2 (/ 59)	20.9 (/59)
Tower-7	6.1 (127/127)	7.0 (/127)	*
Tower-9	23.6 (511/511)	28.0 (/511)	*
8puzzle-1	9.2 (31/31)	33.7 (/ 31)	9.5 (/39)
8puzzle-2	7.0 (32/32)	28.3 (/ 30)	5.5 (/ 48)
TSP-12	6.9 (12/12)	21.1 (/12)	18.9 (/12)
AIPS 1998	PEGG	Alt Alt	
grid-y-1	16.8 (14/14)	17.4 (/14)	17.5 (/14)
gripper-x-5	110 (23/35)	9.9 (/35)	8.0 (/37)
gripper-x-8	520 (35/53)	73 (/48)	25.0 (/53)
log-y-5	30.5 (16/34)	44 (/38)	29.0 (/42)
mprime-1	2.1 (4/6)	722.6 (/ 4)	79.6 (/ 4)
AIPS 2000	PEGG	Alt Alt	
blocks-10-1	6.9 (32/32)	13.3 (/32)	7.1 (/36)
blocks-12-0	9.4 (34/34)	17.0 (/34)	
blocks-16-2	40.9 (56/56)	61.9 (/56)	
logistics-10-0	7.3 (15/ 53)	31.5 (/53)	
logistics-12-1	17.4 (15/75)	80 (/77)	
freecell-2-1	19.1 (6/10)	49 (/12)	
schedule-8-9	297 (5/12)	123 (/15)	
AIPS 2002	PEGG	Alt Alt	
depot-6512	2.1 (14/31)	1.2 (/33)	
depot-1212	79.1 (22/53)	290 (/61)	
driverlog-2-3-6e	80.6 (12/26)	50.9 (/28)	
driverlog-4-4-8	889 (23/38)	461 (/44)	
roverprob1423	15 (9/28)	2.0 (/33)	
roverprob4135	379 (12 / 43)	292 (/45)	
roverprob8271	220 (11 / 39)	300 (/ 45)	
sat-x-5	25.1 (7/22)	3.1 (/25)	
sat-x-9	9.1 (6/35)	5.9 (/ 35)	
ztravel-3-7a	101 (10/23)	77 (/28)	
ztravel-3-8a	15.1 (9/26)	15.4 (/31)	

Table 5: PEGG and a state space planner using variations of the 'adjusted-sum' heuristic

PEGG: bounded PE search, best 20% of search segments visited in each search episode, as ordered by adjusted-sum-u state space heuristic

AltAlt: Lisp version, state space planner with two of the most effective planning graph distance-based heuristics: "adjusum2" and "combo" (combo results are not reported for all problems since adjusum2 produces plans that are more competitive with PEGG in terms of makespan.)

Surprisingly, in the majority of problems PEGG returns a parallel, generally step-optimal plan faster than AltAlt returns its serial plan. (AltAlt cannot construct a plan with parallel actions, however recent work with a highly modified version of AltAlt does, in fact, construct such plans -Nigenda & Kambhampati, 2003). The PEGG plans are also seen to be of comparable length, in terms of number of actions, to the best of the AltAlt plans.

6. Discussion of Results

A distinguishing feature of the EGBG and PEGG planners relative to other planners that exploit the planning graph, is their aggressive use of available memory to learn online from their episodic search experience so as to expedite search in subsequent episodes. Although they each employ a search trace structure to log this experience, the EGBG and PEGG systems differ in both the content and granularity of the search experience they track and the aggressiveness in their use of memory. They also differ in how they confront a common problem faced by learning systems; the utility of learned information versus the cost of storing and accessing it when needed.

Our first efforts focused primarily on using a search trace to learn mutex-related redundancies in the episodic search process. Although the resulting planners, EGBG and me-EGBG, can avoid virtually all redundant mutex checking based on search experience embodied in their PE's, empirically we find that it's a limited class of problems for which this is a winning strategy. The utility of tracking the mutex checking experience during search is a function of the number of times that information is subsequently used. Specifically:

$$6-1)\quad U_{mt}(p) \propto \frac{\sum_{e=1}^{EPS(p)} PE_{visit}(e)}{\sum_{e=1}^{EPS(p)} PE_{add}(e)}$$

where: U_{mt} is the utility of tracking mutex checking experience
p is a planning problem
$EPS(p)$ is the number of search episodes in problem p
$PE_{visit}(e)$ is the number of PE search segments visited in search episode e
$PE_{add}(e)$ is the number of new search segments added to PE in episode e

Thus the payback for EGBG's incurred overhead of tracing consistency-checking experience during search depends on the number of times the sets are revisited relative to the total number of subgoal sets generated (and added to the PE) during the problem run. This characteristic explains the less than 2x speedups observed for me-EGBG on many Table 2 problems. The approach is a handicap for single search episode problems. It is also ineffectual for problems where final search episode search generates a large number of states relative to previous episodes and when the only seed segment(s) are at the top levels of the PE (due to need for bottom-up visitation of the search segments in EGBG's search trace).

The PE can be thought of as a snapshot of the 'regression search reachable' (RS reachable) states for a search episode. That is, once the regression search process generates a state at level k of the planning graph, the state is reachable during search at all higher levels of the graph in future search episodes. Essentially, the search segments in the PE represent not just the RS reachable states, but a

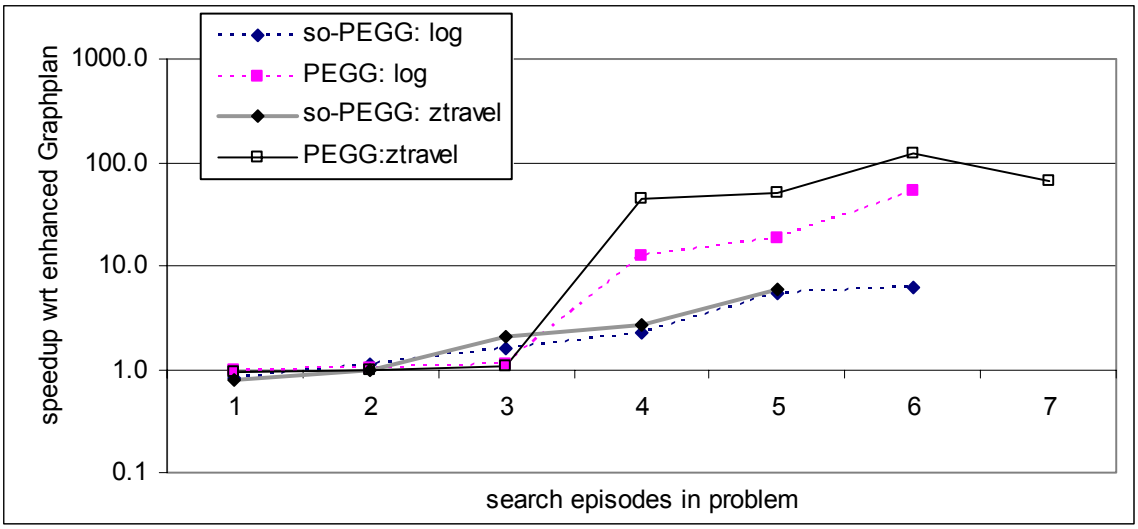

Figure 12: Speedup vs. number of search episodes: Logistics '00 and Zenotravel '02 domains

candidate set of partial plans with each segment's state being the current tail state of such a plan. Table 3 and 5 results indicate that the utility of learning the states that are RS reachable in a given search episode generally outweighs the utility of learning details of the episode's consistency-checking, and require much less memory. Freed from the need to regenerate the RS reachable states in IDA* fashion during each search episode, PEGG can visit such states in any heuristically preferred order.

Tables 2, 3 and 5 shed light on several classes of problems that are problematic for search trace guided search on the planning graph:

1. Domains with high branching factors and operator descriptions that thwart DDB and EBL (e.g. the larger *Schedule*, *Satellite,* and Zenotravel domain problems)
2. Problems with a significant fraction of runtime consumed in planning graph construction. (e.g. *Grid domain,, dlog-2-3-6e, freecell-2-1*)
3. Problems with only one or two search episodes (*grid-y-1, schedule-8-5*)

For problems in the first class, the Graphplan-style CSP assignment search is prone to bogging down after certain PE states are selected for visitation. For those in the second class any search time reduction can be dominated by the large graph construction time (a problem shared by any planner that builds the complete planning graph). Problems in the third class do not give PEGG sufficient opportunity to exploit the PE, since it is built in the first episode (and the first episode PE is typically small) and of no benefit until subsequent episodes. This aspect of PEGG's behavior is illustrated in Figure 12. Here the speedup factors of both so-PEGG and PEGG (under beam search) are plotted for a series of problems ordered according to the number of search episodes that Graphplan would conduct prior to finding a solution. The data was gathered by running the GP-e, so-PEGG, and PEGG planners on two different domains (the Logistics domain from the AIPS-00 planning competition, and the Zenotravel domain from the AIPS-02 competition) and then averaging the speedups observed for problems with the same number of observed search episodes. The downturn for the PEGG/ Ztravel curve at seven episodes is not surprising given that there was only one such problem and there are many factors beyond the number of search episodes that impact solution time. Noting that the speed-

ups are plotted on a logarithmic scale, the power of a search trace given multiple search episode problems is evident. PEGG using beam search handily outperforms so-PEGG for all problems of three or more search episodes, largely because it shortcuts exhaustive search in the intermediate episodes.

There are several avenues for addressing the above-listed limitations of PEGG that we have explored or anticipate investigating. For example, unlike the 'N-best first' state space planner reported in Table 4, PEGG enforces the user-specified limit on state f-values *only* when selecting PE search segments to visit. Once a search segment is chosen for visitation, Graphplan-style regression search on the state goals continues until either a solution is found or all sub-branches fail. A more greedy approach would be to also apply the heuristic bound during this regression search. That is, we could backtrack whenever a state is generated that exceeds the f-value threshold applied to search segments before they are visited. This translates the Greedy Best First Search (GBFS) algorithm employed by HSP-r (Bonet & Geffner, 1999) for state space search, into a form of hill-climbing search on the planning graph.

Experimentally we find that when PEGG is adapted to enforce the PE state f-value limit during its regression search, improvements are unpredictable at best. Speedups of up to a factor of 100 were observed in a few cases (all logistics problems) but in many cases runtimes increased or search failed entirely within the time limit. In addition, the quality (make-span) of the returned solutions suffered across a broad range of problems. There are two factors that may explain this result: 1) PEGG's regression search is greatly expedited by DDB and EBL, but the regressed conflict set that they rely on is undefined when the regression search space below a state is not fully explored, such as when an f-value limit is enforced. Without conducting such search there is no informed basis for returning anything other than the full set of subgoals in the state, which essentially forces search towards chronological backtracking. 2) Assessing an f-value for a newly generated state to compare against an f-value bound that is based on states generated in previous episodes is problematic. This is because the heuristic values of PE states that determine the f-value bound have been increased by PEGG's use of search experience to improve h-value estimates (Section 5.1.2).

Degradation of solution quality as we shift PEGG closer to a greedy search approach may be an indicator that PEGG's ability to return step-optimal plans (as evidenced by Table 3 results) is rooted in its interleaving of best-state selection from the PE with Graphplan-style depth-first search on the state's subgoals.

7. Related Work

We focus here on related or alternative strategies for employing search heuristics in planning, generating parallel plans, or making use of memory to expedite search. Related work pertaining to some of the search techniques, efficiencies, and data structures that enable EGBG and PEGG to successfully employ a search trace were cited as they arose above and are not further considered here.[17]

As noted in Section 2.2, a shortcoming of IDA* search (and Graphplan) is its inadequate use of available memory: The only information carried over from one iteration to the next is the upper bound on the f-value. Exploitation of a search trace directly addresses this shortcoming by serving as a memory of the *states* in the visited search space of the previous episode in order to reduce redun-

[17] Support methodologies include memory efficient bi-partite planning graph models, explanation based learning and dependency directed backtracking in the context of planning graph search, variable and value ordering strategies, and the evolution and extraction of distance-based heuristics from the planning graph.

dant regeneration. In this respect PEGG's search is closely related to methods such as MREC (Sen, Anup & Bagchi, 1989), MA*, and SMA* (Russell, 1992) which lie in the middle ground between the memory intensive A* and IDA*'s scant use of memory. A central concern for these algorithms is using a prescribed amount of available memory as efficiently as possible. Like EGBG and PEGG, they retain as much of their search experience as memory permits to avoid repeating and regenerating nodes, and depend on a heuristic to order the nodes in memory for visitation. Unlike our search trace based algorithms though, all three of the above algorithms backup a deleted node's f-value to the parent node. This ensures the deleted branch is not re-expanded until no other more promising node remains on the open list. We have not implemented this extended memory management in PEGG (though it would be straight-forward to do so) primarily because, at least under beam search, PEGG has seldom confronted PE-related memory limitations.

EGBG and PEGG are the first planners to directly interleave the CSP and state space views in problem search, but there are related approaches that synthesize different views of the planning problem. The Blackbox system (Kautz & Selman 1999) constructs the planning graph but instead of exploiting its CSP nature, it is converted into a SAT encoding after each extension and a k-step solution is sought. GP-CSP (Do & Kambhampati, 2000), similarly alternates between extending a planning graph and converting it, but it transforms the graph into CSP format and seeks to satisfy the constraint set in each search phase.

The beam search concept is employed in the context of propositional satisfiability in GSAT (Selman, Levesque, & Mitchell, 1992) and is an option for the Blackbox planner (Kautz & Selman, 1999). For these systems greedy local search is conducted by assessing in each episode, the n-best 'flips' of variable values in a randomly generated truth assignment (Where the best flips are those that lead to the greatest number of satisfied clauses). If n flips fail to find a solution, GSAT restarts with a new random variable assignment and again tries the n-best flips. There are several important differences relative to PEGG's visitation of the n-best search trace states. The search trace captures the state aspect engendered by Graphplan's regression search on problem goals and as such, PEGG exploits reachability information implicit in its planning graph. In conducting their search on a purely propositional level, SAT solvers can leverage a global view of the problem constraints but cannot exploit state-space information. Whereas GSAT (and Blackbox) do not improve their performance based on the experience from one n-best search episode to the next, PEGG learns in a variety of modes; improving its heuristic estimate for the states visited, reordering the state goals based on prior search experience, and memorizing the most general no-goods based on its use of EBL.

Like PEGG, the LPG system (Gerevini & Serina, 2002) heavily exploits the structure of the planning graph, leverages a variety of heuristics to expedite search, and generates parallel plans. However, LPG conducts greedy local search in a space composed of subgraphs on a given length planning graph, while PEGG combines a state space view of its search experience with Graphplan's CSP-style search on the graph itself. LPG does not systematically search the planning graph before heuristically moving to extend it, so the guarantee of step-optimality is forfeited. PEGG can operate either in a step-optimal mode or in modes that trade off optimality for speed to varying degrees.

We are currently investigating an interesting parallel to LPG's ability to simultaneously consider candidate partial plans of different lengths. In principle, there is nothing that prevents PEGG from *simultaneously* considering a given PE search segment S_n, in terms of its heuristic rankings when it's transposed onto various levels of the planning graph. This is tantamount to simultaneously consider-

ing which of an arbitrary number of candidate partial plans of different implied lengths to extend first (each such partial plan having S_n as its tail state). The search trace again proves to be very useful in this regard as any state it contains can be transposed up any desired number of levels -subject to the ability to extend the planning graph as needed- and have its heuristics re-evaluated at each level. Referring back to Figure 4, after the first search episode pictured (top), the YJ state in the PE could be expanded into multiple distinct states by transposing it up from graph level 5 to levels 6, 7, or higher, and heuristically evaluating it at each level. These graph-level indexed instances of YJ can now be simultaneously compared. Ideally we'd like to move directly to visiting YJ at planning graph level 7, since at that point it becomes a plan segment for this problem (bottom graph of Figure 4). If our secondary heuristic can discriminate between the solution potential for a state at the sequential levels it can be transposed to, we should have an effective means for further shortcutting Graphplan's level-by-level search process. The flux adjunct is likely to be one key to boosting the sensitivity of a distance-based heuristic in this regard.

Generating and assessing an arbitrarily large number of graph-level transposed instances of PE states would be prohibitive in terms of memory requirements if we had to store multiple versions of the PE. However we can simply store any level-specific heuristic information in the search segments of a single PE as values indexed to their associated planning graph levels. Challenging issues include such things as the range of plan lengths to be considered at one time and the potential for plans with steps consisting entirely of 'persists' actions.

We haven't examined PEGG in the context of real-time planning here, but its use of the search trace reflects some of the flavor of the real-time search methods, such as LRTA* (Korf, 1990) and variants such as B-LRTA* (Bonet, Loerincs, & Geffner, 1997), -a variant that applies a distance-based heuristic oriented to planning problems. Real-time search algorithms interleave search and execution, performing an action after a limited local search. LRTA* employs a search heuristic that is based on finding a less-than-optimal solution then improving the heuristic estimate over a series of iterations. It associates an h-value with every state to estimate the goal distance of the state (similar to the h-values of A*). It always first updates the h-value of the current state and then uses the h-values of the successors to move to the successor believed to be on a minimum-cost path from the current state to the goal. Unlike traditional search methods, it can not only act in real-time but also amortize learning over consecutive planning episodes if it solves the same planning task repeatedly. This allows it to find a sub-optimal plan fast and then improve the plan until it converges on a minimum-cost plan.

Like LRTA*, the PEGG search process iteratively improves the h-value estimates of the states it has generated until it determines an optimal make-span plan. Unlike LRTA*, PEGG doesn't actually find a sub-optimal plan first. Instead it converges on a minimum-cost plan by either exhaustively extending all candidate partial plans of monotonically increasing length (so-PEGG) or extending only the most promising candidates according to its secondary heuristic (PEGG with beam search). A real-time version of PEGG more closely related to LRTA* might be based on the method described above, in which search segments are simultaneously transposed onto multiple planning graph levels. In this mode PEGG would be biased to search quickly for a plan of any length, and then search in anytime fashion on progressively shorter length planning graphs for lower cost plans.

This methodology is of direct relevance to work we have reported elsewhere on "multi-PEGG" (Zimmerman & Kambhampati, 2002; Zimmerman 2003), a version of PEGG that operates in an anytime fashion, seeking to optimize over multiple plan quality criteria. Currently multi-PEGG first re-

turns the optimal make-span plan, and then exploits the search trace in a novel way to efficiently stream plans that monotonically improve in terms of other quality metrics. As discussed in that paper, an important step away from multi-PEGG's bias towards the make-span plan quality metric would be just such a modification. Co-mingling versions of the same state transposed onto multiple planning graph levels would enable the planner to concurrently consider for visitation candidate search segments that might be seed segments for latent plans of various lengths.

8. Conclusions

We have investigated and presented a family of methods that make efficient use of available memory to learn from different aspects of Graphplan's iterative search episodes in order to expedite search in subsequent episodes. The motivation, design, and performance of four different planners that build and exploit a search trace are described. The methods differ significantly in either the information content of their trace or the manner in which they leverage it. However, in all cases the high-level impact is to transform the IDA* nature of Graphplan's search by capturing some aspect of the search experience in the first episode and using it to guide search in subsequent episodes, dynamically updating it along the way.

The EGBG planners employ a more aggressive mode of tracing search experience than the PEGG planners. They track and use the action assignment consistency checking performed during search on a subgoal set (state) to minimize the effort expended when the state is next visited. The EGBG approach was found to be memory intensive, motivating the incorporation of a variety of techniques from the planning and CSP fields which, apart from their well-known speedup benefits, are shown to have a dramatic impact on search trace and planning graph memory demands. The resulting planner, me-EGBG, is frequently two orders of magnitude faster than either standard Graphplan or EGBG and for problems it can handle, it is generally the fastest of the guaranteed step-optimal approaches we investigated. In comparisons to GP-e, a version of Graphplan enhanced with the same space saving and speedup techniques, me-EGBG solves problems on average 5 times faster.

The PEGG planners adopt a more skeletal search trace, a design more conducive to informed traversal of the search space. Ultimately this proves to be a more powerful approach to exploiting the episodic search experience. We adapt distance-based, state space heuristics to support informed traversal of the states implicit in the search trace and describe a metric we call "flux" which effectively focuses search on states worth visiting. This flux measure is sensitive to the potential for a search trace state to seed new search branches as it is transposed to higher planning graph levels. We also describe some new techniques that leverage the search experience captured in the search trace and demonstrate their effectiveness.

The so-PEGG planner, like me-EGBG, produces guaranteed optimal parallel plans and similarly averages a 5x speedup over GP-e. Its greatly reduced memory demands allow so-PEGG to handles all but one of the 16 problems for which me-EGBG exceeds available memory. More compelling evidence of the speedup potential for a search trace guided planner is provided by PEGG under beam search. Since it no longer exhaustively searches the planning graph in each episode, PEGG sacrifices the guarantee of returning an optimal make-span plan. Nonetheless, even under beam search limited to just the best 20% of PE states in each episode, PEGG returns the step-optimal plan in almost 90% of the test bed problems and comes within a few steps of optimal in the others. It does so at speedups

ranging to almost two orders of magnitude above GP-e, and quite competitively with a modern state space planner (which finds only serial plans).

The code for the PEGG planners (including GP-e) with instructions for running them in various modes is available for download at http://rakaposhi.eas.asu.edu/pegg.html

Acknowledgements

This research was improved by many discussions with Binh Minh Do, XuanLong Nguyen, Romeo Sanchez Nigenda and William Cushing. Thanks also to David Smith and the anonymous reviewers, whose copious suggestions greatly improved the presentation of this paper. This research is supported in part by the NSF grants IRI-9801676 and IIS-0308139, DARPA AASERT Grant DAAH04-96-1-0247 and the NASA grants NAG2-1461 and NCC-1225.

Appendix A: The EGBG Planner

The insight behind EGBG's use of a search trace is based on the characterization of Graphplan's search given at the beginning of Section 3.1 and some entailed observations:

Observation A-1) The intra-level CSP-style search process conducted by Graphplan on a set of propositions (subgoals) S, at planning graph level $k+1$ in episode $n+1$ is *identical* to the search process on S at level k in episode n IF:

1. All mutexes between pairs of actions that are establishers of propositions of S at level k remain mutex for level $k+1$. (this concerns dynamic mutexes; static mutexes persist by definition)
2. There are no actions establishing a proposition of S at level $k+1$ that were not also present at level k.

Observation A-2) The trace of Graphplan's search during episode $n+1$, on a set of goals G, at planning graph level $m+1$, is identical to its episode n search at level m IF:

1. The two conditions of observation A-1 hold for *every subgoal set* (state) generated by Graphplan in the episode $n+1$ regression search on G.
2. For every subgoal set S at planning graph level j in search episode n for which there was a matching level j memo, there exists an equivalent memo at level $j+1$ when S is generated in episode $n+1$. Conversely, for every subgoal set S at level j in search episode n for which no matching level j memo existed at the time it was generated, there is also no matching memo at level $j+1$ at the time S is generated in episode $n+1$.

Now, suppose we have a search trace of all states (including no-good states) generated by Graphplan's regression search on the problem goals from planning graph level m in episode n. If that search failed to extract a solution from the m-length planning graph (i.e. reach the initial state), then a necessary condition to extract a solution from the $m+1$ length graph is that one or more of the conditions of observations in A-1 or A-2 fails to hold for the states in the episode n search trace.

With observations A-1 and A-2 in mind, we can exploit the search trace in a new episode in a sound and complete manner by focusing search effort on the only three situations that could lead to a solution: 1) under state variables with newly extended value ranges (i.e. search segment goals that have at least one new establishing action at their newly associated graph level), 2) at points in the previous search episode that backtracked due to violation of a 'dynamic' constraint (i.e. two actions that

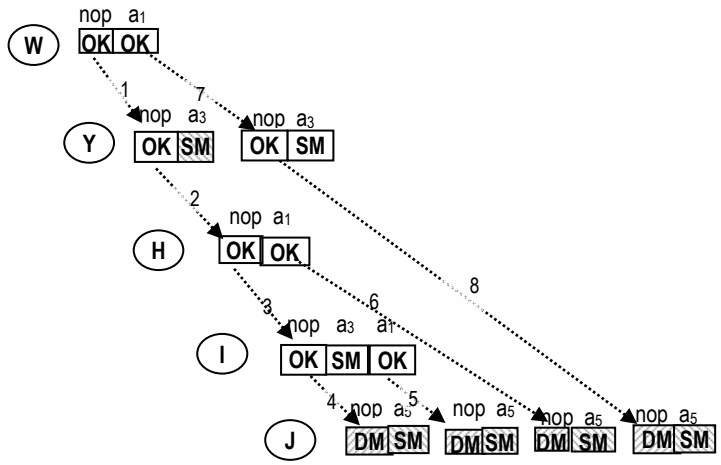

Figure A1: Bit vector representation of the search trace for the WYHIJ state in Figure 3.
Semantics: OK –> assigned (no action conflicts) SM –> action is static mutex with previous assign
DM –> action is dynamic mutex with previous assign
NG –> a no-conflict action results in a no-good state at lower graph level

were dynamic mutex), and 3) at states that matched a cached memo in episode *n.* All other assignment and mutex checking operations involved in satisfying a set of subgoals are static across search episodes.

We experimented with several search trace designs for capturing key decision points. The design adopted for EGBG employs an ordered sequence of bit vectors, where each vector contains the results of Graphplan's CSP-style action assignment process related to a given subgoal in a search segment. Efficient action assignment replay is possible with a trace that uses vectors of two-bit tags to represent four possible assignment outcomes: 1) dynamic mutex, 2) static mutex, 3) no conflict, and 4) a complete, consistent set of assignments that is rejected at the next level due to a memoized no-good. Figure A1 illustrates how a sequence of eight such bit vectors can be used to capture the search experience for the search segment with state goals WYHIJ from our Figure 3 *Alpha* problem. Here the propositional goals (the variables) appear to the left of the sets of bit vectors (depicted as segmented bars) which encode the outcome of all possible action assignment (the values). Each possible establishing action for a goal appears above a bit vector tag.

The numbered edges reflect the order in which the trace vectors are initially created when the first goal action is tried. Note that whenever a candidate action for a goal is conflict free with respect to previously assigned actions (indicated by the OK in the figure), action checking for the goal is suspended, the process jumps to the next goal, and a new bit vector is initialized for this goal's possible establishers. The edge numbering also reflects the order in which the vectors are popped from the search segment trace list when the segment is revisited in the next episode. For this scheme to work, the bit vectors must be pushed onto the search segment trace list after all actions for a goal are tried, in the reverse of the numbered edge order. Long edges that skip over one or more goals indicate that those goals are already established by previously assigned actions.

As long as the *order* of actions appearing under the "establishers" list for a planning graph proposition remains constant, the bit vectors can be used to replay the search in the next episode on the next higher planning graph level. The graph building routine for EGBG enforces this constraint.

The EGBG Algorithm

The high-level EGBG algorithm is given in Figure A2. As for Graphplan, search on the planning graph occurs only after it has been extended to the level where all problem goals first appear with no binary mutex conditions. (the call to *find_1st_level_with_goals*). The first search episode is conducted in Graphplan fashion except that the *assign_goals* routines of Figure A3 create search segments to hold the states and trace information generated during the regression search process. The necessary trace information for a search segment is captured in trace vectors as described above. These segments are stored in the PE structure indexed according to the level at which they where generated (where the current highest planning graph level corresponds to 0 and contains the problem goals).

Subsequent to the first episode, *EGBG_plan* enters an outer loop that employs the PE to conduct successive episodes (Referred to as "search trace guided"). The search strategy alternates between the selection and visitation of a promising state from the trace of previous experience (*select_searchseg_from_PE* routine), with a focused CSP-type search on the state's subgoals (the *replay_trace_goals* and *assign_goals* routines of Figures A3 and A4).

For each episode, an inner loop visits the PE search segments in level-by-level, bottom-up fashion (for the reasons discussed in Section 3). The *extend_plangraph* routine called only when a state being visited corresponds to a level beyond the current graph length.

The *replay_trace_goals* routine is the counterpart to Graphplan's *assign_goals* routine, except that it avoids the latter's full-blown mutex checking by stepping through the trace vectors that captured previous search experience for a given state. Unlike *assign*_goals, it does not branch to any child states already contained in the PE. The conditional checking of the trace vectors against establishing actions initiates new search by calling *assign*_goals under two conditions: 1) when dynamic mutexes from previous episodes no longer hold 2) when new establishing actions appear for a subgoal (These are tried after all other establishers are replayed.) When a dynamic mutex no longer holds or a new establishing action is considered the trace vector is modified accordingly and EGBG resumes Graphplan's CSP-style search, adding new trace vectors to the search segment in the process.

```
EGBG-PLAN ( Ops, Init, Goals)   /* {Ops,Init,Goals} is a planning problem */
  /* build plangraph, PG, until level n where goals first occur in non-mutex state*/
 Let PG ← find_1st_level_with_goals( Ops, Init, Goals )
 if PG reached level-off and goals are not present in non-mutex state then Return FAIL
 Let n be the number of levels in PG
 Let SS0 be a new search segment with fields:
              goals←Goals, parent←'root, PE-level← 0, parent-actions← {} trace← {}
 Let PE ← pilot explanation structure with fields to hold search segments at each plangraph level
 PE[0] ←{SS0}   /* 0 is top level of PE */
  /* Conduct Graphplan-style backward search on the n-length plangraph, store trace in the PE...*/
 Let search-reslt ← assign_goals(Goals, {}, n, SS0, PG, PE)
 if search-reslt is a search segment /* Success… */
   then Plan ← extract plan actions from the ancestors linked to search-reslt
        Return Plan
  else /* No n-length solution possible ...use the PE to search for a longer length solution */
    loop forever
      n ← n+1
      loop for pe-lev ranging from the number of deepest level of PE to 0 (top level)
        let k ← planning graph level associated with PE level pe-lev
             = n – pe-lev   /* ..essentially translates the PE up one planning graph level */
        if pe-lev = 0 then PG ← extend_plangraph(PG)   /*.. must extend plangraph at this point */
        loop for search segments in PE[pe-lev]
             SS ← select_searchseg_from_PE[pe-lev]
             SSassigns ← SS<trace> /* get ordered, goal-by-goal trace vectors from search segment */
             SS<trace> ← {} /* Clear the search segment trace vectors field */
             if SS<goals> ∈ memos(k, PG) /*check for nogoods at level k */
                then these goals match a nogood at this level, loop to next search segment
              else /* use SS trace to avoid redundant search effort on SS goals.. */
                search-reslt ← replay_trace_goals (SS<goals>, {}, k, SSassigns, SS, PG, PE)
                if search-reslt is a search segment
                   then Plan ← extract plan actions from the ancestors linked to search-reslt
                        Return Plan
                  else Add SS<goals> to memos(k, PG)   /*memoize nogood */
        end loop (search segments)
      end loop (PE levels)
    end-loop (PG level)
end
```

Figure A2: EGBG planner top level algorithm

```
Conduct Graphplan-style search on a subgoal set at planning graph level k
   arguments> G: goals still to be assigned, A: actions already assigned, k: PG level,
              SS1: search segment, PG: planning graph,  PE: pilot explanation (search trace)
ASSIGN_GOALS (G, A, k, SS1, PG, PE)
 if G is empty or k = 0 (the initial state) then Return SS1   /* Success */
  else   /* there are goals left to satisfy…*/
    Let g-assigns ={}    /*  trace vector will hold ordered action assignment tags */
    Let g be a goal selected from G
    Let Ag = actions from PG level k that support g, ordered by value-ordering heuristic
    loop for  act∈   Ag
     Let search-reslt = {}
     if  an action in A is dynamic mutex with act then append 'dm' tag to g-assigns
       else if  an action in A is static mutex with act then append 'sm' tag to g-assigns
       else    /* act has no conflict with actions already in A */
          if  G is empty
            then    /* done with G goals, setup  for search at next lower level */
               search-reslt ← assign_next_level_goals (A ∪{act}, k, SS1, PG, PE)
               if search-reslt is a 'nogood' then append 'ng' tag to g-assigns
                 else append 'ok' tag to g-assigns    /*  search occurred at lower level */
           else    /* search continues at this level with the next goal… */
              append 'ok' tag to g-assigns
              search-reslt ← assign_goals (G-{g}, A ∪{act}, k, SS1, PG, PE)
          end-if
          if search-reslt is a search segment  then  Return search-reslt   /* Success */
            /* else we loop and  try another action…*/
      end-if
    end loop (actions)
    push g-assigns into trace field of SS1    /*add trace data to search segment */
    Return nil  /* no solution found */
 end-if
end

Setup for search on graph level k-1 given that the actions in A satisfy the goals of SS1 at level k
ASSIGN_NEXT_LEVEL_GOALS (A, k, SS1, PG, PE)
 Let nextgoals ← regress SS1 goals over assigned actions in A
 if nextgoals ∈ memos at PG level k-1 then Return 'nogood'   /* backtrack on nogood goals */
   else  /* … initiate search on next lower PG level*/
      Let SS2 be a new search segment with fields:
                  goals←nextgoals,  parent←SS1,  parent-actions←A,  trace←{}
      Add SS2 to the PE at level: (maximum PG level) – (k-1)
      Let search-reslt ← assign_goals (nextgoals, {}, k-1, SS2, PG, PE)
      if search-reslt is nil    /* search at level k-1 failed */
        then  Add nextgoals to memos to level k-1 of PG    /*memoize nogood */
      Return search-reslt
 end-if
end

ASSIGN_NEW_ACTIONS (G, A, Ag, g, g-assigns, k, SS, PG, PE)
 /* Routine is essentially the same as assign_goals, except it attempts to satisfy goal g actions
with  only the 'new' actions (i.e. those first appearing in the most recent plangraph extension */
```

Figure A3: EGBG's non-guided regression search algorithm

```
Regression search using search trace (PE) replay
   arguments> G: goals still to be assigned, A: actions already assigned, k: PG level,
              SS1: search segment, PG: planning graph,  PE: pilot explanation (search trace)
REPLAY_TRACE_GOALS  (G, A, k, SS1, PG, PE)
 if G is empty   /* all SS1 goals in this branch were successfully assigned during  last episode...*/
   then Return   /* ..  continue with level k replay, ignoring search replay at next lower level  */
  else   /* there are goals left to satisfy…*/
    Let g ← select goal from G
    Let g-assigns ← pop the front trace vector from SS<trace>
    Let Ag ←set of actions from level k of PG that support g
    /* Now replay assignments for g  from previous episode, rechecking only those that may have changed..*/
    loop for  tag ∈  g-assigns
       Let search-reslt ←{},    Let act ← pop action from Ag
       if tag = 'ok'   /* act had no conflict with actions in A during last episode.. go to next goal…  */
          then  search-reslt ← replay_trace_goals (G-{g}, A ⋃{act}, k, SSassigns, SS1, PG, PE)
        else-if  tag ='ng' then loop  /* it's the last action assign at this level & act was not mutex in the last
          episode --So the next-level regressed goals reside in a child search segment that was already visited */
        else if  tag = 'sm' then loop  /* act was static mutex with action in A last episode */
        else tag = 'dm'   /* act was dynamic mutex with action  in A last episode …retest it...*/
            if dynamic mutex persists then loop
              else change tag to 'ok' in g-assigns vector    /* act is no longer mutex with A actions */
                 if G-{g} is not empty    /*  resume backward search at this level… */
                    then search-reslt ← assign_goals (G-{g}, A ⋃{act}, k, SS1, PG,PE)
                   else   /* no goals left to satisfy in SS1, setup for search at lower level */
                     search-reslt ← assign_next_level_goals (A ⋃{act}, k, SS1, PG, PE)
                     if search-reslt ='nogood' then change g-assigns vector tag to 'ng'
       end-if
       if search-reslt is a search segment  then  Return search-reslt  (Success)
          else loop (check next action)
    end-loop   /* all establishment possibilities from prior episode tried ..Now check for new actions */
    if Ag still contains actions   /* they are new actions establishing g at this level .. attempt to assign */
      then search-reslt ← assign_new_actions(G, A, Ag, g, g-assigns, k, SS1, PG, PE)
          if search-reslt is a search segment  then  Return search-reslt  /* Success */
     else push g-assigns → SS1<trace>
    Return nil   /* no solution found in search stemming from SS1 goals */
 end-if
end
```

Figure A4: EGBG's search-trace guided algorithm

Appendix B: Exploiting CSP Speedup Methods to Reduce Memory Demands

Background and implementation details are provided here for the six techniques from the planning and CSP fields which proved to be key to controlling memory demands in our search trace based planners. They are variable ordering, value ordering, explanation based learning (EBL), dependency directed backtracking (DDB), domain preprocessing and invariant analysis, and replacing the redundant multi-level planning graph with a bi-partite version.

Domain preprocessing and invariant analysis:

The speedups attainable through preprocessing of domain and problem specifications are well documented (Fox & Long, 1998; Gerevini & Schubert, 1996). Static analysis prior to the planning process can be used to infer certain invariant conditions implicit in the domain theory and/or problem specification. The domain preprocessing for me-EGBG and PEGG is fairly basic, focusing on identification and extraction of invariants in action descriptions, typing constructs, and subsequent rewrite of the domain in a form that is efficiently handled by the planning graph build routines. Our implementation discriminates between static (or permanent) mutex relations and dynamic mutex relations (in which a mutex condition may eventually relax) between actions and proposition pairs. This information is used to both expedite graph construction and during me-EGBG's 'replay' of action assignments when a search segment is visited.

Domain preprocessing can significantly reduce memory requirements to the extent that it identifies propositions that do not need to be explicitly represented in each level of the graph. (Examples of terms that can be extracted from action preconditions -and hence do not get explicitly represented in planning graph levels- include the (*SMALLER ?X ?Y)* term in the MOVE action of the 'towers of Hanoi' domain and typing terms such as *(AUTO ?X)* and *(PLACE ?Y)* in logistics domains.) This benefit is further compounded in EGBG and PEGG since propositions that can be removed from action preconditions directly reduce the size of the subgoal sets generated during the regression search episodes, and hence the size of the search trace.

Bi-partite planning graph:

The original Graphplan maintains the level-by-level action, proposition, and mutex information in distinct structures for each level, thereby duplicating -often many times over- the information contained in previous levels. This multi-level planning graph can be efficiently represented as an indexed two-part structure and finite differencing techniques employed to focus on only those aspects of the graph structure that can possibly change during extension. This leads to more rapid construction of a more concise planning graph (Fox & Long 1998; Smith & Weld, 1998).

For me-EGBG and PEGG, the bi-partite graph offers a benefit beyond the reduced memory demands and faster graph construction time; the PE transposition process described in section 3.1 is reduced to simply incrementing each search segment's graph level index. This is not straightforward with the multi-level graph built by Graphplan, since each proposition (and action) referenced in the search segments is a unique data structure in itself.

Explanation Based Learning and Dependency Directed Backtracking:

The application of explanation based learning (EBL) and dependency directed backtracking (DDB) were investigated in a preliminary way in (Zimmerman & Kambhampati, 1999), where the primary interest was in their speedup benefits. The techniques were shown to result in modest speedups on

several small problems but the complexity of integrating them with the maintenance of the PE replay vectors limited the size of problem that could be handled. We have since succeeded in implementing a more robust version of these methods, and results reported here reflect that.

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, 1998). 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. An approach that implements these complimentary techniques for Graphplan is reported in (Kambhampati, 2000) where speedups ranged from ~2x for 'blocksworld' problems to ~100x for 'ferry' domain problems. We refer to that study for a full description of EBL/DDB in a Graphplan context, but note here some aspects that are particularly relevant for me-EGBG and PEGG.

As for conflict directed back-jumping (Prosser, 1993), the failure explanations are compactly represented in terms of "conflict sets" that identify the specific action/goal assignments that gave rise to backtracking. This liberates the search from chronological backtracking, allowing it to jump back to the most recent variable taking part in the conflict set. When all attempts to satisfy a set of subgoals (a state) fail, the conflict set that is regressed back represents a useful 'minimal' no-good for memoization. (See the PEGG algorithm in Figures 8 and 9 for a depiction of this process.) This conflict set memo is usually shorter and hence more general than the one generated and stored by standard Graphplan. Additionally, an EBL-augmented Graphplan generally requires less memory for memo caches.

Less obvious than their speedup benefit perhaps, is the role EBL and DDB often play in dramatically reducing the memory footprint of the pilot explanation. Together EBL and DDB shortcut the search process by steering it away from areas of the search space that are provably devoid of solutions. Search trace memory demands decrease proportionally.

Both me-EGBG and PEGG have been outfitted with EBL/DDB for all non-PE directed Graphplan-style search. me-EGBG however, does *not* use EBL/DDB in the 'replay' of the action assignment results for a PE search segment due to the complexity of having to retract parts of assignment vectors whenever the conflict set in a new episode entails a new replay order.

Value and Variable Ordering:

Value and variable ordering are also well known speedup methods for CSP solvers. In the context of Graphplan's regression search on a given planning graph level k, the variables are the regressed subgoals and the values are the possible actions that can give these propositions at level k of the graph. In their original paper, Blum and Furst (1997) argue that variable and value ordering heuristics are not particularly useful in improving Graphplan, mainly because exhaustive search is required in the levels before the solution bearing level anyway. Nonetheless, the impact of dynamic variable ordering (DVO) on Graphplan performance was examined in (Kambhampati, 2000), and modest speedups were achieved using the standard CSP technique of selecting for assignment the subgoal ('variable') that has the least number of remaining establishers ('values'). More impressive results are reported in a later study (Nguyen & Kambhampati, 2000) where distance-based heuristics rooted in the planning graph were exploited to order both subgoals and goal establishers. In this configuration, Graphplan exhibits speedups ranging from 1.3 to over 100x, depending on the particular heuristic and problem.

For this study we fix variable ordering according to the 'adjusted sum' heuristic and value ordering according the 'set level' heuristic, as we found the combination to be reasonably robust across the range of our test bed problems. These heuristics are described in Section 5 where they are used to direct the traversal of the PE states is discussed. Section 4.1 describes the highly problem-dependent performance of distance-based variable and value ordering for our search trace-based planners.

The manner in which EGBG/PEGG builds and maintains the planning graph and search trace structures actually reduces the cost of variable and value ordering. The default order in which Graphplan considers establishers (values) for satisfying a proposition (variable) at a given level is set by the order in which they appear in the planning graph structure. During graph construction in me-EGBG and PEGG we can set this order to correspond to the desired value ordering heuristic, so that the ordering is only computed once. For its part, the PE that is constructed during search can record the heuristically-best ordering of each regression state's goals, so that this variable ordering is also done only once for the given state. This contrasts with versions of Graphplan that have been outfitted with variable and value ordering (Kambhampati, 2000) where the ordering is reassessed each time a state is regenerated in successive search episodes.

References

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

Bonet, B., Loerincs, G., & Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97.*

Bonet, B. & Geffner, H. (1999). Planning as heuristic search: New results. In *Proceedings of ECP-99.*

Do, M.B. & Kambhampati, S. (2000). Solving Planning-Graph by compiling it into CSP. In *Proceedings of AIPS-00.*

Fox, M., & Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9, 317-371.

Frost, D. & Dechter, R. (1994). In search of best constraint satisfaction search. In *Proceedings of AAAI-94.*

Gerevini , A., & Schubert, L. (1996). Accelerating Partial Order Planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research* 5, 95-137.

Gerevini, A. & Serina, I., (2002). LPG: A planner based on local search for planning graphs with action costs. In *Proceedings of AIPS-02.*

Haslum, P., & Geffner, H. (2000). Admissible Heuristics for Optimal Planning. In *Proceedings. of AIPS-00*.

Hoffman, J. (2001) A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. Technical Report No. 133, Albert Ludwigs University.

Kambhampati, S. (1998). On the relations between Intelligent Backtracking and Failure-driven Explanation Based Learning in Constraint Satisfaction and Planning. *Artificial Intelligence,* 105(1-2).

Kambhampati, S. (2000). Planning Graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of Artificial Intelligence Research*, 12, 1-34.

Kambhampati, S. & Sanchez, R. (2000). Distance-based Goal-ordering heuristics for Graphplan. In *Proceedings of AIPS-00.*

Kambhampati, S., Parker, E., & Lambrecht, E. (1997). Understanding and extending Graphplan. In *Proceedings of ECP-97*.

Kautz, H. & Selman, B. (1996). Pushing the envelope: Planning, prepositional logic and stochastic search. *In Proceedings of AAAI-96.*

Kautz, H. & Selman, B. (1999). Unifying SAT-based and Graph-based Planning. In *Proceedings of IJCAI-99,* Vol 1.

Koehler, D., Nebel, B., Hoffman, J., & Dimopoulos, Y., (1997). Extending planning graphs to an ADL subset. In *Proceedings of ECP-97*, 273-285.

Korf, R. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence,* 27(1), 97-109.

Korf, R. (1990). Real-time heuristic search. *Artificial Intelligence,* 42, 189-211.

Long, D. & Fox, M. (1999). Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research*, 10, 87-115.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90.*

McDermott, D. (1999). Using regression graphs to control search in planning. *Artificial Intelligence*, 109(1-2), 111-160.

Nigenda, R., & Kambhampati, S. (2003). AltAlt^p: Online Parallelization of Plans with Heuristic State Search. *Journal of Artificial Intelligence Research*, 19, 631-657.

Nguyen, X. & Kambhampati, S. (2000). Extracting effective and admissible state space heuristics from the planning graph. In *Proceedings of AAAI-00.*

Prosser, P. (1993). Domain filtering can degrade intelligent backtracking search. In *Proceedings of IJCAI-9*3.

Russell, S.J., (1992). Efficient memory-bounded search methods. In *Proceedings of ECAI 92*.

Sen, A.K., & Bagchi, A., (1989). Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of IJCAI-89.*

Selman, B, Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92.*

Smith, D., Weld, D. (1998). Incremental Graphplan. Technical Report 98-09-06. Univ. of Wash.

Srivastava, B., Nguyen, X., Kambhampati, S., Do, M., Nambiar, U. Nie, Z., Nigenda, R., Zimmerman, T. (2001). AltAlt: Combining Graphplan and Heuristic State Search. In *AI Magazine*, 22(3), American Association for Artificial Intelligence, Fall 2001.

Zimmerman, T. (2003). Exploiting memory in the search for high quality plans on the planning graph. PhD dissertation, Arizona State University.

Zimmerman, T. & Kambhampati, S. (1999). Exploiting Symmetry in the Planning-graph via Explanation-Guided Search. In *Proceedings of AAAI-99*.

Zimmerman, T., Kambhampati, S. (2002). Generating parallel plans satisfying multiple criteria in anytime fashion. In *Proceedings of workshop on Planning and Scheduling with Multiple Criteria*, *AIPS-02*.

Zimmerman, T. & Kambhampati, S. (2003). Using available memory to transform Graphplan's search. Poster paper in *Proceedings of IJCAI-03*.