Using Memory to Transform Search on the Planning Graph

Terry Zimmerman zim@asu.edu
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 of these heuristics targeted specifically at planning graph search. Guided by such a heuristic, the step-optimal version of 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 state-of-the-art 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, to search speedup techniques such as variable and value ordering, dependency-directed backtracking, and explanation based learning. In spite of these advances, Graphplan has ceded the lead in planning speed to a variety of heuristic-guided planners (Bonet and Geffner, 1999, Nguyen and Kambhampati, 2000, Gerevini and 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 an family of approaches that retain attractive features of Graphplan's IDA* search, such as rapid generation of parallel action steps and the ability to find step optimal plans, while surmounting some of its 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. The methodology remains rooted in iterative search on the planning graph but greatly expedites this search by employing available memory to build and maintain a concise search trace. Depending on the particular approach used, the search trace can allow the planner employing it to 1) successfully avoid much of the redundant search effort, 2) learn from its iterative search experience so as to improve its heuristics and the constraints embodied in the planning graph, and 3) realize 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.

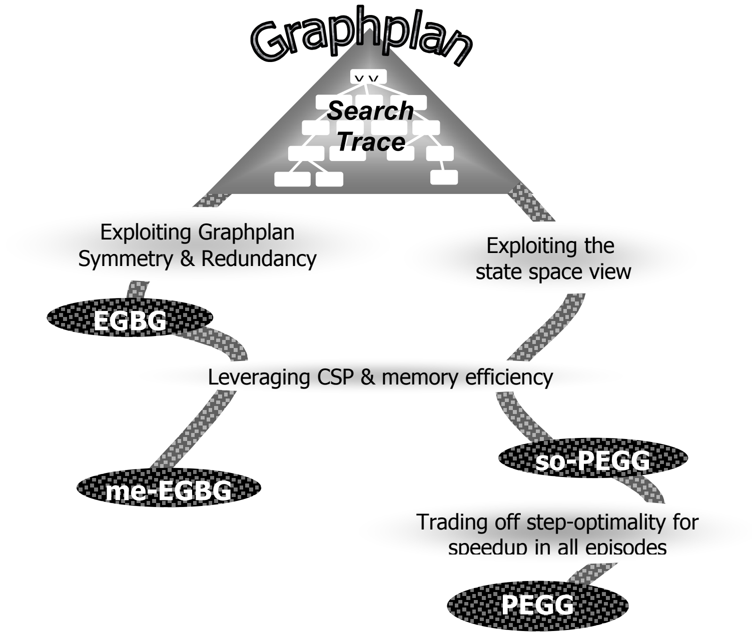
Graphplan
Search Trace
Exploiting Graphplan Symmetry & Redundancy
Exploiting the state space view
EGBG
Leveraging CSP & memory efficiency
so-PEGG
me-EGBG
Trading off step-optimality for speedup in all episodes
PEGG

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

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 of a search trace approach to expediting search on the planning graph, we explore several variations 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 the two genres of search trace we worked with;

- *left track:* The EGBG planners (Explanation Guided Backward search for Graphplan) employ a more comprehensive search trace focused on minimizing redundant search effort.
- *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 and 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 is shown to be effective in a range of smaller problems but memory constraints 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 secondly explore some middle ground in terms of memory usage by augmenting the underlying planner 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. This attention to memory efficiency in the me-EGBG system markedly improves the speed and capabilities of the planner, but a variety of problems remain that lie beyond the planner's reach due to memory constraints. This motivates a shift to the greatly pared down search trace employed by the PEGG planners that trades off minimization of redundant search in exchange for a much smaller memory footprint.

It's an interesting upshot of employing available memory to build and maintain a search trace that it provides us with a state space view of what is, essentially, Graphplan's CSP-oriented search space. During each iteration, Graphplan uses a depth-first strategy to build a consistent set of actions (values) satisfying a set of subgoals (variables) at each level of the planning graph. The proposition set that is sub-goaled on at each level in the regression search process essentially constitutes an incomplete 'state'. However, existing planners that conduct search directly on the planning graph largely ignore such states, save for the rudimentary learning of invalid states in the form of 'no-goods'. A compelling strategy suggested by the search trace and exploited by the PEGG planners is to adopt a global view of the regression search state-space generated in episode *n*, and to select particular regions of that space in episode *n+1* for early expansion. Having chosen a heuristically desirable state, PEGG is then able to continue its expansion iteratively in Graphplan's CSP-style, depth-first fashion in search of a parallel, step-optimal plan. The first step can exploit powerful 'distance-based' heuristics (that have been key to the success of the fastest serial state-space planners) as a secondary heuristic in Graphplan's search, while the second step employs a variety of CSP speed-up techniques to shortcut the search below a selected state. In the process PEGG also learns in unique fashion from its iterative search experience, both augmenting the mutex constraints in the planning graph and improving the heuristic evaluation functions used to select states for expansion.

This third approach is implemented first in a planner called so-PEGG ('step-optimal PEGG', Zimmerman and Kambhampati, 2003). Beyond its greatly reduced memory demands, so-PEGG is distinguished from the EGBG track planners in its adaptation of the 'distance-based' heuristics that power some of the current generation of state-space planners (Bonet and Geffner, 1999, Nguyen and Kambhampati, 2000, Hoffman, 2001). These heuristics exhibit significant shortcomings when applied to the task of traversing the search trace and this leads us to develop a specialized measure of the potential for a state in the trace to seed *new* search on the planning graph beyond those states generated in previous search episodes. The metric, which we term 'flux', significantly improves the effectiveness of a distance-based heuristic in identifying the most promising states to visit in the search trace.

With this capability, so-PEGG achieves an important degree of independence from Graphplan's strict depth-first search process, and exploits it to outperform even a highly enhanced version of Graphplan by up to two orders of magnitude in terms of speed. It does so while still maintaining the guarantee that the returned solution will be a step-optimal plan.

The last avenue we explore is to adopt 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. The flux metric is shown to be effective not only in augmenting the state-ordering secondary heuristic, but as a filter for setting a threshold below which a search trace state can be skipped over even though it might appear promising based on the distance-based heuristic. The implemented system (PEGG, Zimmerman and Kambhampati, 2003), realizes 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 state-of-the-art state space planner using similar heuristics. In adopting beam search, PEGG necessarily sacrifices the *guarantee* of step-optimality. Nonetheless, our empirical results indicate that the secondary heuristics are quite effective in ensuring that the quality of returned plans, in terms of makespan, is 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 and Fox, 1999), Blackbox (Kautz and Selman, 1999), IPP (Koehler, et. al., 1997), AltAlt (Nguyen and Kambhampati, 2000), LPG (Gerevini and Serina, 2002). The planning systems described here share that memory overhead of course, but interestingly, we have found that search trace memory demands associated with 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 and 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 (More complete discussion of the techniques is provided in Appendix B). 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 state-of-the-art, heuristic serial state-space planner. Section 6 contains a discussion of our findings and Section 7 compares this work to related research. Finally, Section 8 presents our conclusions.

2 Background & Motivation: Planning graphs and the nature of direct graph search

Here we outline the Graphplan algorithm and discuss characteristics 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.

2.1 Construction and search on a planning graph

The Graphplan algorithm (Blum & Furst, 1997) consists of two interleaved phases – a forward phase, where a data structure called ``planning graph" is incrementally extended, and a backward phase where the planning graph is searched to extract a valid plan. The planning graph consists of two alternating structures, called proposition lists and action lists. At the bottom of Figure 2 is depicted a simple domain we will call the *Alpha* domain that will be used throughout this study. The figure displays four action and proposition levels of the planning graph engendered by this domain. We start with the initial state as the zeroth level proposition list. Given a k-level planning graph, the

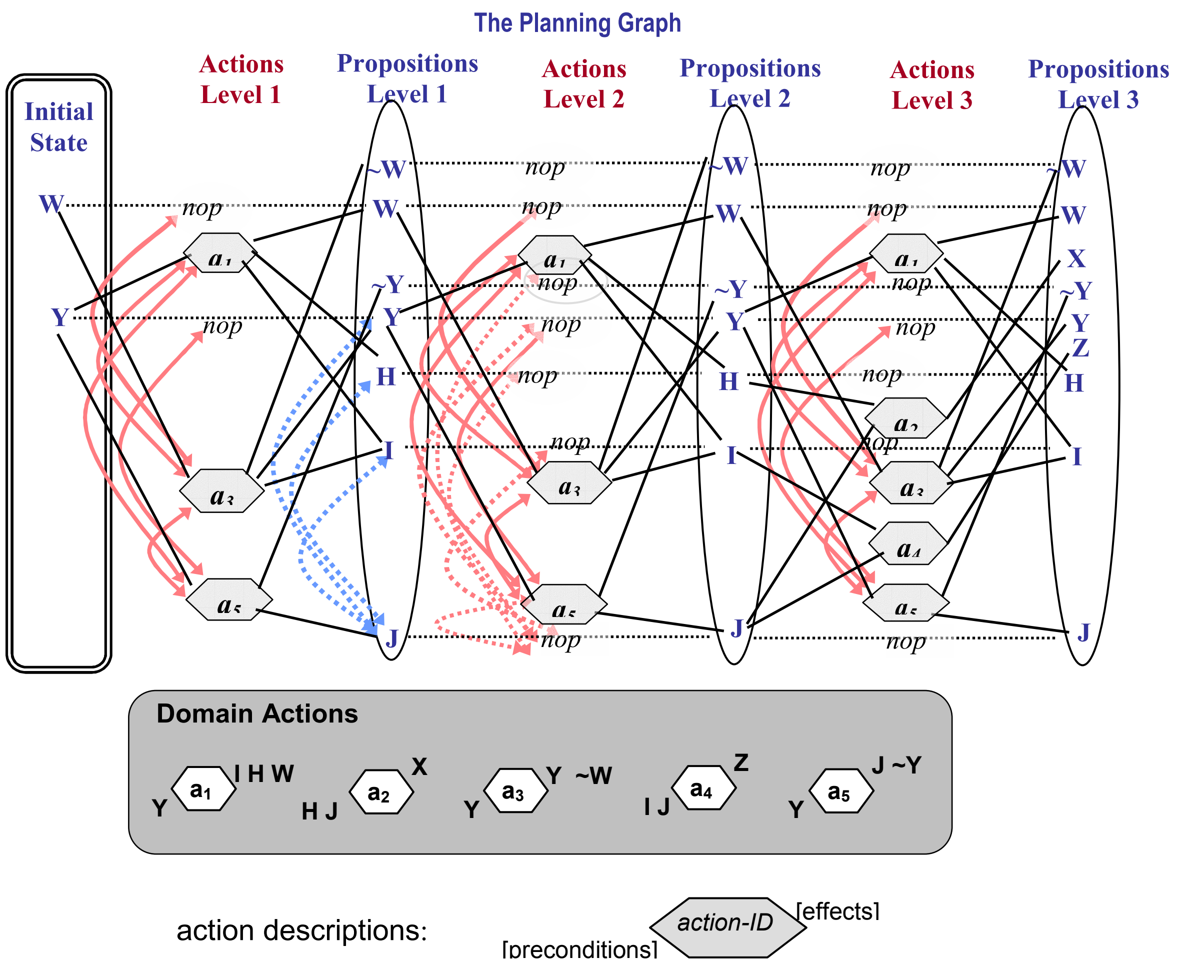

Figure 2: Planning graph representation for three levels in the Alpha domain

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, we introduce "no operation" actions (abbreviated as "nop" in this paper's figures), one for each condition in the k^{th} level proposition list. 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 level k proposition list, and their effects in level *k+1* proposition list.

Planning graph construction also involves computation and propagation of binary "mutex" constraints. In Figure 2, the arcs denote the 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, 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 k+1 are mutex if they are statically interfering ("static mutex") or if one of the propositions/preconditions supporting the first action is mutually exclusive with one of the propositions supporting the second action. (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 (they are the negation of each other) or dynamic mutex (all actions establishing one proposition are mutex with all actions establishing the other). To reduce cluttering in Figure 2 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 a given level are the variables and the actions that establish a proposition are the values. 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 (corresponding to 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 and 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 there will be no changes in actions, propositions, or mutex conditions.

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

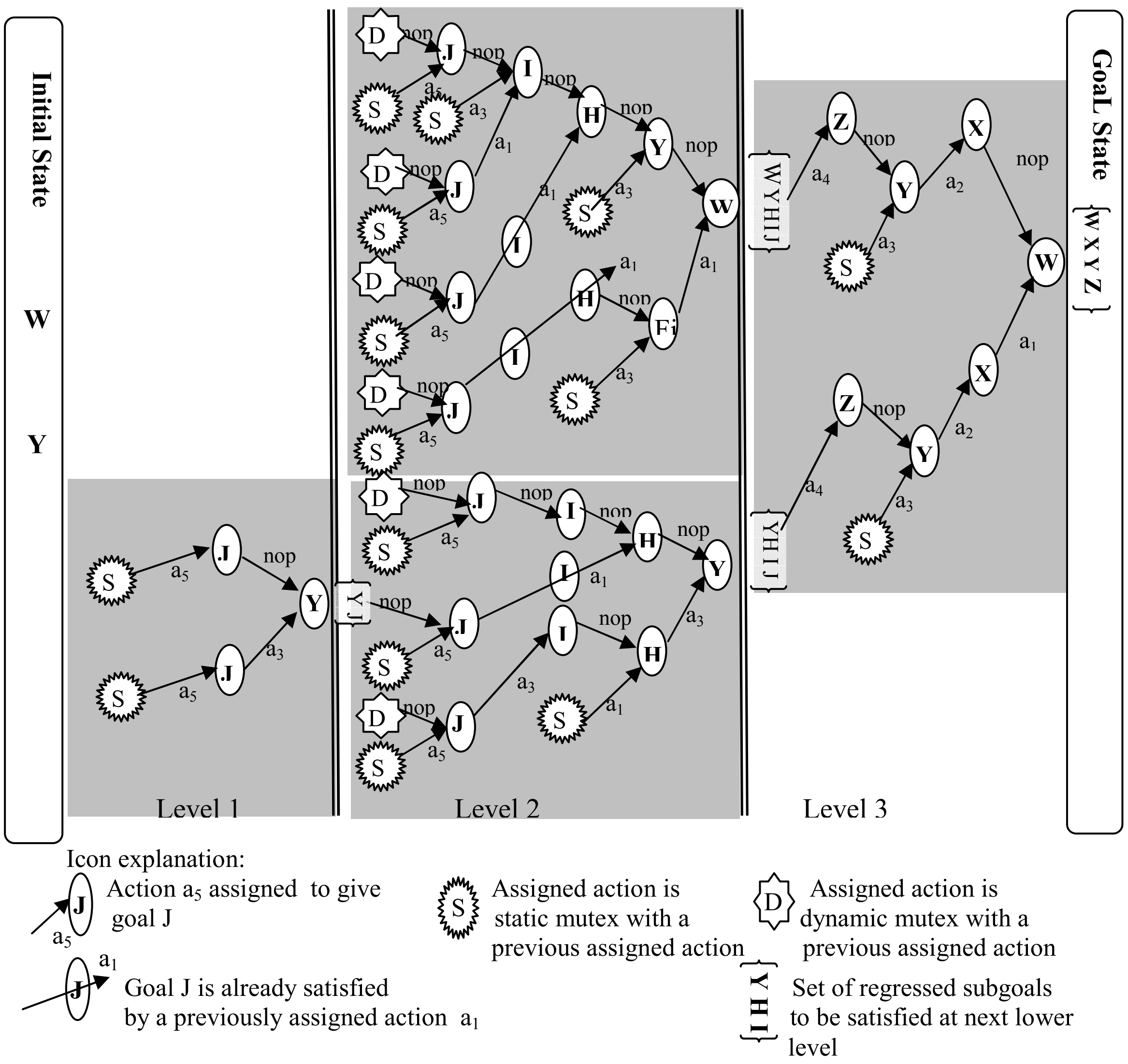
Initial State
W
Y
GoaL State
W X Y Z
Level 1
Level 2
Level 3
Y J
W Y H I J
Y H I J
Icon explanation:
Action a5 assigned to give goal J
Assigned action is static mutex with a previous assigned action
Assigned action is dynamic mutex with a previous assigned action
Goal J is already satisfied by a previously assigned action a1
Set of regressed subgoals to be satisfied at next lower level

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

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'. It is important to note that although the graph need not be further extended, finding a solution may require continuing for many more episodes the process of adding a new static level and conducting regression search on the problem goals.

Like many fielded CSP solvers, Graphplan's search process benefits from a simple form of no-good 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 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.

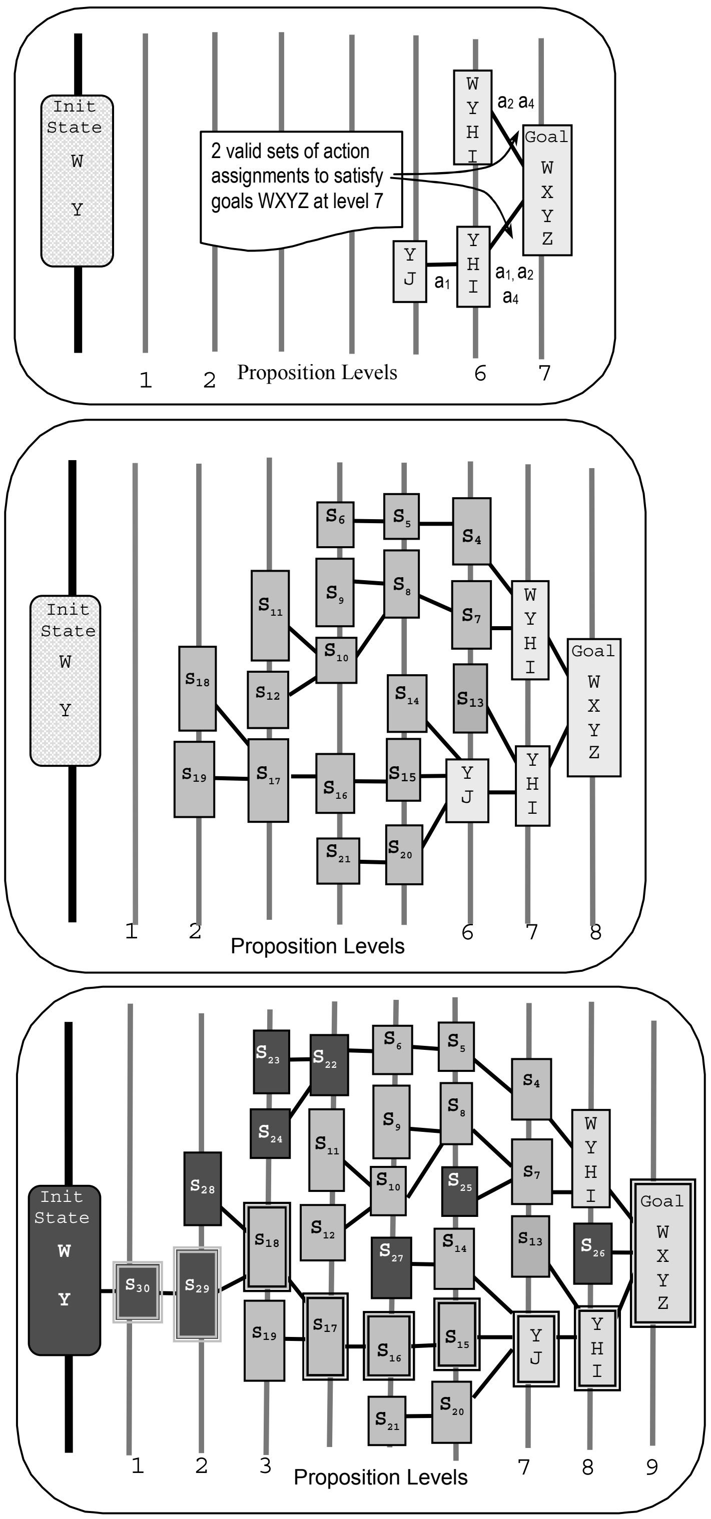
Init State W Y
2 valid sets of action assignments to satisfy goals WXYZ at level 7
W Y H I
a2 a4
Goal W X Y Z
Y J
a1
Y H I
a1, a2 a4
1
2
Proposition Levels
6
7
S6 S5 S4 S11 S9 S8 S7 S10 S18 S12 S14 S13 S19 S17 S16 S15 S21 S20
1 2 6 7 8
S23 S22 S24 S28 S25 S27 S26 S30 S29
1 2 3 7 8 9

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

2.2 Graphplan as state space search

From a more abstract perspective, Graphplan's search process can be viewed as regression state space search from the problem goals to the initial state. The 'states' that are generated and expanded in this case 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 and 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 of Graphplan's search is illustrated in Figure 4, where the top graph casts the CSP-style search trace of 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]

Once we recognize the state-space aspect of Graphplan's search, its connection to IDA* search becomes more apparent. First noted and briefly discussed in (Bonet and Geffner, 1999), we highlight and expand upon this relationship here. The connection is based on recognizing three correspondences between the algorithms:

1. Graphplan's episodic search process in which all nodes generated in the previous episode are regenerated in the new episode (possibly along with some new nodes), corresponds to IDA*'s iterative search. Here the Graphplan nodes are the 'states' (sets of subgoals) that result when its regression search on a given

[2] Here instead of the first search episode beginning on planning graph level 3, as in Figure 3, we have conjured up a hypothetical problem in which it begins on level 7. This will be useful in illustrating the search trace discussed in the next section.

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 heuristic f-value for node-states;

   $f = g + h$ where:

   h is the distance in terms of associated planning graph levels between the state generated in Graphplan's regression search and the initial state[3]

   g is the cost of reaching the node-state from the goal state in terms of number of CSP epochs (i.e. the numerical difference between the highest planning graph level and the state's level).

For our purposes, perhaps the 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 the fact 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 and 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 arises from the observation made above regarding the f-value for the node-states it generates; 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] Bonet and Geffner define the Graphplan *h-value* somewhat differently; they define h_G as the first level at which the goals of a state appear non-mutex and have not been memoized. The definition given here (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.

3 Efficient use of a search trace to guide planning graph search

The search space Graphplan explores 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 similarity (i.e. redundancy) in the search space for successive search 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 "re-play" of the search conducted at the previous level *k* with certain well-defined extensions. More specifically, essentially *every* set of subgoals reached in the backward search of episode *n*, starting at level *k*, will be generated again 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 the 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 these propositions is mutex. Note that in the middle Figure 4 graph depicting the next backward search episode, 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 depicts a hypothetical variation of the Alpha domain and the problem detailed in Figures 2 and 3, all states created beyond the first episode are only labeled with state numbers suggesting 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 and 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

[4] Strictly speaking, this is not always the case due to the impact of Graphplan's memorizing 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. It simplifies visualization of the symmetry across Graphplan's search episodes to neglect these relatively rare exceptions to the above characterization of the search process.

the tactic adopted by EGBG. We provide here only an overview of the EGBG approach to maximal reduction of redundant search effort, referring the interested reader to Appendix A for details. The EGBG search trace 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*.
- 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 at a level 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.

The above characteristics suggest an approach for expediting search in episode *n*+1 given that we have 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 episode *n* search that are provably unchanged in episode *n*+1, and focus search effort on only features that may have evolved. If previous search failed to extract a solution from the k-length planning graph (i.e. reach the initial state), then the only way a solution can be extracted from the *k+1* length graph is if one or more of these conditions holds:

1. The dynamic mutex condition between some pair of actions that 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 a action that establishes it and first appears in level *k+1.*
3. A regression state (subgoal set) generated in episode *n* at level *k* that matched a cached memo at that level, no longer matches a memo when it is generated at level *k+1* in episode *n+1.*

The discussion in Appendix A formalizes these conditions. For each one that does *not* hold, the backward search must be resumed under the search parameters corresponding to the backtrack point 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 illustrating the search process:

Search segment: This is essentially a state, in particular 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), that is, 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 trace of the most recent results of the action consistency-checking process during the attempt to satisfy the subgoals. When attempting to assign a goal-achieving action 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 bit vectors for efficiency.)

A search segment therefore represents a state plus some path information, but we often use 'search segment' and 'state' interchangeably. All the boxes in Figure 4 (whether the state goals are explicitly shown or not) can be viewed as search segments. The procedure used by EGBG to build and use these search segments is discussed in Appendix A.

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. (See Appendix A for description of the algorithm.)

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-

[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.

type search on the state's subgoals. In the latter mode, having chosen a state to visit, it 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 candidate 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.

The focus of our discussion of EGBG's bi-modal algorithm revolves around the second mode (minimizing redundant search effort once a state has been chosen for visitation), but it will become apparent when we report on the PEGG's use of the search trace that the first mode of EGBG's search process, the selection of a promising state from the trace, has the greater potential for dramatic efficiency increases. Since the PE can be viewed as encapsulating a search space of states, we may no longer be restricted to the (non-informed) depth-first nature of Graphplan's search process and have the freedom to traverse the states in any preferred order. Unfortunately, EGBG incurs a high overhead associated with visiting the search segments in any order *other than* bottom up (in terms of PE levels). When an ancestor of any state represented in the PE is visited before the state itself, EGBG's search process will *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.

Top-down visitation of the segments in the PE levels is the degenerate mode. This search process will essentially mimic 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 will avoid all the search effort the Graphplan search process 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 and Geffner, 1999, Nguyen and 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 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 and 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. The Tower-of-Hanoi problems fall into this classification.

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

[6] The values differ from those published in 1999 because problems were re-run on the same machine used for other experiments in this study. They also reflect some changes in the tracking of statistics.

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 in the problem column give number of time steps / number of actions respectively in the solution. Search backtracks and mutex checks performed during the search are shown. "Size of PE" is the pilot explanation size in terms of the final number of search segments. Stnd Graphplan: Lisp version by Smith and Peot. Run times in cpu seconds on a 900 MHz, Pentium III, 384 MB RAM, running Allegro common lisp.

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 significant 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.

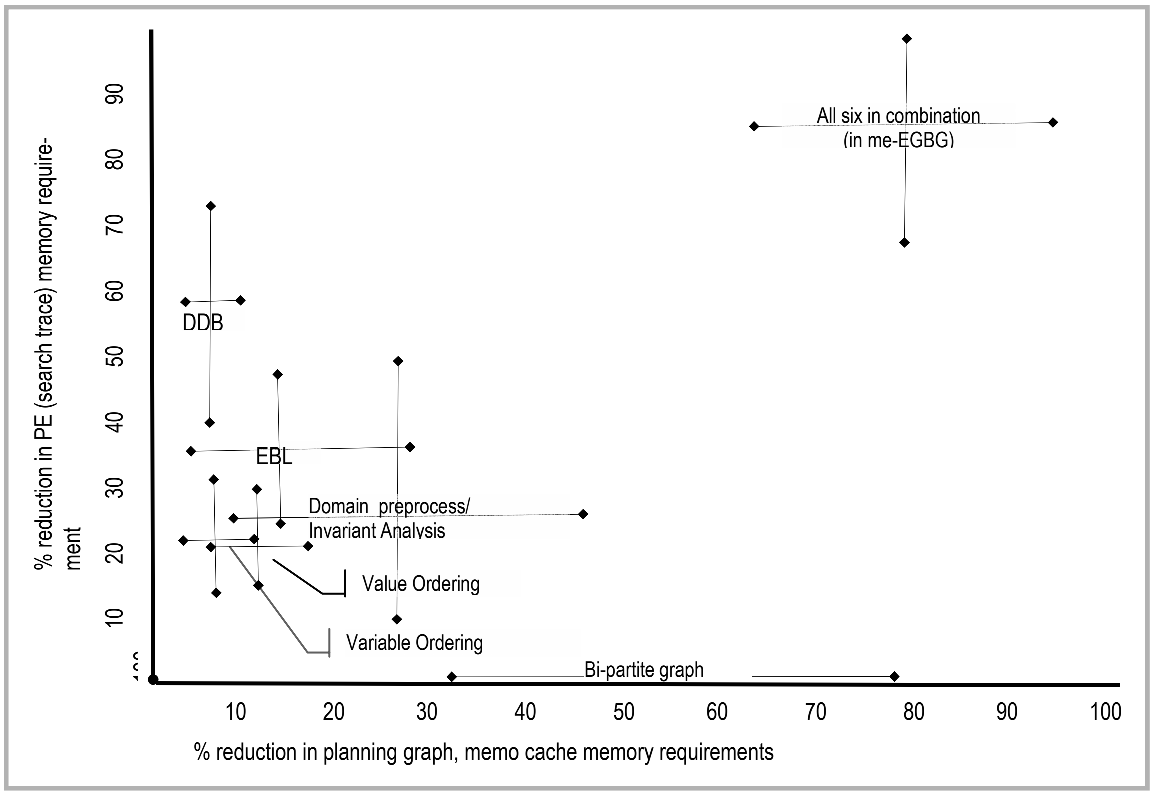

Figure 5: Memory demand impact along two dimensions for six memory reduction/ speedup techniques. Plots for each applied independently and as a suite (within EGBG).

4.1 Impact of enhancements on EGBG memory demands

There are two major modes in which the first six techniques impact memory demand in the context of 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 have nonetheless chosen to partition along these lines to facilitate a clear comparison of each technique's impact on the search trace that distinguishes our planning approach.

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

The Figure 5 plot 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 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 36 benchmark problems in a wide range of domains, including problems from all three AIPS planning competitions held to date. The problems were selected to satisfy several objectives; a subset that both standard Graphplan and EGBG could

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

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 36 problems, standard Graphplan solves 11, the original EGBG solves 12, GP-e solves 28, and me-EGBG solves 27. Wherever me-EGBG and GP-e solve the same problem, me-EGBG is faster by up to a factor of 62x, and averages >5x speedup. *Standard* Graphplan (on the eleven 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. As indicated by the columns reporting the size of the PE (in terms of search segments at the time the problem is solved), me-EGBG generates and retains in its trace up to 100x fewer states than the first version. This translates into a much broader reach for me-EGBG; it exhausts memory on only 4 problems compared to 19 for the first version of EGBG. Nonetheless, GP-e solves three problems on which me-EGBG fails in 30 minutes due to search trace memory demands (For two problems, GP-e fails to find a solution where the latter succeeds in the allotted time runtime window).

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. Evidence of this is apparent in examining the final search episodes for the problems of Table 2; in the great majority, the PE is found to contain a seed segment (a state from which regression search will reach the initial state) within the deepest two or three PE levels. This supports the intuition discussed in the previous section and suggests that the advantage of a low *h-value* bias as observed for heuristic state-space planners (Bonet and Geffner, 1999, Nguyen and Kambhampati, 2000) translates to search on the planning graph.

Problem	Graphplan		EGBG		me-EGBG		SPEEDUP
(steps/actions)	cpu sec				(memory efficient EGBG)		(me-EGBG vs. GP-e)
	Stnd.	GP-e (enhanced)	cpu sec	size of PE	cpu sec	size of PE	
bw-large-B (18/18)	126	11.4	79	7919	9.2	2090	1.2x
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
att-log-b (11/79)	*s*	*s*	*pe*		*s*		~
gripper-8 (15/23)	125	14.1	88	9790	12.9	2313	1.1x
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*		*(pe)*
8puzzle-2 (30/30)	304	48.3	*pe*		26.9	10392	1.8x
TSP-12 (12/12)	*s*	454	*pe*		21.0	7155	21.6x
AIPS 1998	Graphplan	GP-e	EGBG		me-EGBG		
grid-y-1 (14/14)	388	16.7	393		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*		*(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		
blocks-10-1 (32/32)	*s*	101.4	*pe*		20.3	6788	5.0x
blocks-12-0 (34/34)	*s*	30.6	*pe*		21.5	3220	1.42x
logistics-10-0 (15/56)	*s*	30.0	*s*		16.6	1115	1.81x
logistics-11-0 (13/56)	*s*	78.3	*pe*		10.0	1377	7.8x
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-8-9 (5/12)	*pg*	175	*pg*		164	230	1.1x
AIPS 2002	Graphplan	GP-e	EGBG		me-EGBG		
depot-1212 *(22/55)*	*pg*	*s*	*pg*		*s*		~
depot-6512 (10/26)	239	5.1	219	4272	4.1	456	1.25x
depot-7654a (10/28)	*s*	32.5	*s*		14.8		2.2x
driverlog-2-3-6a (10/24)	1280	2.8	807	1569	1.0	232	3.8x
driverlog-2-3-6b (7/20)	*s*	27.5	1199	2103	3.9	401	7x
roverprob1425 (10/32)	*s*	21.9	979	10028	12.5	2840	1.8x
roverprob1423 (9/30)	*s*	170	*pe*		84.4	4009	2.5x
ztravel-3-8a (7/25)	*s*	972	*pe*		15.6	1353	62x
ztravel-3-8b (6/22)	*s*	11.0	991	3773	10.2	1353	1.1x

Table 2: Search for step-optimal plans: Comparison of EGBG and memory efficient EGBG with standard, and enhanced Graphplan.

'Stnd Graphplan': Lisp version by Smith and Peot. GP-e and me-EGBG use the Section 4 efficiency methods.
"Size of PE" is 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

Times given in cpu seconds on a 900 MHz, Pentium III, 384 MB RAM, running Allegro common lisp. Letter suffix added to some competition problem names discriminate between different problems with identical names. Numbers in parentheses next to the problem names list the # of time steps and # of actions respectively in the Graphplan solution

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

1. The intra-segment action assignment trace vectors that allow EGBG to avoid redundant search effort 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 search segments in the PE 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). When the only seed segment in the PE is the topmost search segment, bottom-up visitation of the PE states is more costly than Graphplan's top-down approach. A more flexible and informed traversal order seems warranted.

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). To the extent that the overhead associated with building its search trace can be reduced, so too can the hit EGBG takes on such problems compared to, for example, Graphplan. 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 and Geffner, 1999, Nguyen and Kambhampati, 2000, Gerevini and 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 confront more costly bookkeeping and high memory management overhead.[8] More informed traversal of the state-space view of Graphplan's search space is taken up next, where we argue that this is perhaps *the* key advantage afforded by a trace of search on the planning graph.

5 Tracing at the state level and 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 (Pilot Explanation Guided Graphplan) planners employing this search trace -both the so-PEGG (the step-optimal version) planner 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 that the detailed mutex-checking information contained in the bit vectors of the former is replaced by heuristic value(s) used to determine visitation order for the latter.

[8] The state visitation strategy employed by EGBG can be made somewhat more informed by employing state space heuristics to order (and reorder) the search segments *within* each PE level prior to each episode. Such limited reordering of the prior search process only moderately increases the regeneration of states already in the PE. An effective heuristic biases search within a given PE level towards any seed segments that may be present, but the process is still constrained to exhaustively search the segments in the level before moving to any state at a higher level. Empirically we find this gives only modest improvement in EGBG performance and have not reported such results here.

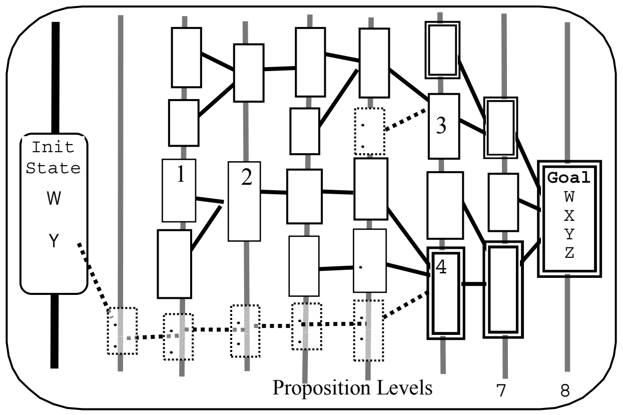

Figure 6. The PE for the final search episode of a hypothetical problem. Search segments in the PE at the onset of search appear in solid lines, double lined boxes represent plan segments, dashed lined boxes are states newly generated in regression search during the episode. Visitation order as dictated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

The principal tactic taken by the PEGG planners is to employ available memory to transform Graphplan's iterative deepening depth-first search into iterative expansion of a select set of states that can be traversed in any desired order. They derive their greatest strength from the global view of the search space visited in previous episodes that is captured in the search trace.

Figure 6 illustrates this advantage by depicting a small hypothetical search trace in the final search episode. Here search segments in the PE at the onset of the episode appear in solid lines and all plan segments (i.e. states that could be extended to find 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 for this problem 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).

The so-PEGG planner guarantees, that returned plans are step-optimal by visiting every search segment in the PE during each search episode (comparable to Graphplan's exhaustive search on a given length graph). 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 number and size of search episodes, the effort required to conduct exhaustive search in even the intermediate episodes becomes prohibitive. The planner we refer to simply as PEGG employs beam search to fully exploit the search trace heuristics in all intermediate search episodes as well. 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 as well as 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 detailing how we addressed these issues, we first present some results that provide perspective on the relative effectiveness of the Graphplan-related planners we discuss in this paper.

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

Problem	**Graphplan** cpu sec (steps/acts) **Stnd.**	**GP-e** (enhanced)	**me-EGBG** cpu sec (steps/acts)	**so-PEGG** heur:istic: adjsum-flux cpu sec (steps/acts)	**PEGG** heur: adjsum-flux cpu sec (steps/acts)	**Speedup** (PEGG vs. GP-e)
bw-large-B	194.8	11.4 (18/18)	9.2	7.0	4.9 (18/18)	2.3x
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
rocket-ext-a	*s*	3.5 (7/36)	1.8	2.8 (7/34)	1.1 (7/34)	3.2x
att-log-a	*s*	31.8 (11/79)	7.2	2.6 (11/72)	2.2 (11/62)	14.5x
att-log-b	*s*	*s*	*pe*	*s*	21.6 (13/64)	> 83x
Gripper-8	*s*	14.0 (15/23)	12.9	16.6	5.5 (15/23)	2.5x
Gripper-15	*s*	*s*	*pe*	347.5	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)*	21.0	7.2	8.9 (12/12)	51x
AIPS 1998	Stnd GP	GP-e	me-EGBG	so-PEGG	PEGG	
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	
blocks-10-1	*s*	95.4 (32/32)	20.3	18.7	6.9 (32/32)	13.8x
blocks-12-0	~	26.6 (34/34)	21.5	23.0	9.4 (34/34)	2.8x
blocks-16-2	*s*	*s*	*pe*	*s*	58.7 (54/54)	> 31 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 (15/74)	> 2.7x
freecell-2-1	*pg*	98.0 (6/10)	*pe*	75	52.9 (6/10)	1.9x
freecell-3-5	*pg*	1885 (7/16)	*pe*	*s*	101 (7/17)	18.7x
schedule-8-9	*pg*	173 (5/12)	164	170	155 (5/12)	1.1x
AIPS 2002	Stnd GP	GP-e	me-EGBG	so-PEGG	PEGG	
depot-6512	239	5.1 (14/31)	4.1	5.0	2.1 (14/31)	2.4x
depot-1212	*s*	*s (22/55)*	*s*	*s*	127 (22/56)	> 14x
depot-7654a	*s*	32.5 (10/28)	14.8	12.9	13.2 (10/26)	2.7x
driverlog-2-3-6e	*s*	166 (12/28)	3.9	2.2	66.6 (12/26)	2.5x
driverlog-4-4-8	*s*	*s*	*pe*	*s*	889 **(23/38)**	> 2x
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	444 (11 / 39)	> 4x
ztravel-3-7a	*s*	*s*	*pe*	1434 (10/23)	222 **(11/24)**	> 8x
ztravel-3-8a	*s*	972 (7/25)	15.6	11.2	3.1 **(9/26)**	314x

Table 3. so-PEGG and PEGG comparison to Graphplan, GP-e, and me-EGBG

GP-e: Graphplan enhanced with bi-level PG, domain preprocessing, EBL/DDB, value/var. ordering
me-EGBG: memory efficient EGBG (bi-level PG, domain preprocessing, EBL/DDB, etc.)
so-PEGG: step-optimal, search via the PE, segments ordered by adjsum-flux heuristic.
PEGG: bounded PE search, best 50 segments visited, as ordered by adjsum-flux heuristic.
Parentheses adjacent to cpu time give (# of steps / # of actions) in the solution returned.
Allegro Lisp platform, runtimes (excl. gc time) on Pentium 900 mhz, 384 M RAM

'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 also 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 Table 3 results the following parameter settings were used based on good performance *on average* across a variety of domains and problems:

- o Secondary heuristic for visiting states:
  adjsum-flux with $w_0=1$ (eqn 5-3) $w_1 = 1$, $w2 = 0.1$ (eqn 5-4)
- o Beam search: visit the 50 best (lowest f-value) search segments per search episode
- o Filtering: only search segments with flux > 0 are visited. (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. Of the 16 problems for which me-EGBG exceeds available memory due to the size of the PE, only one pushes that limit for so-PEGG. Seven of these problems are actually solved by so-PEGG while the remainder exceed the time limit during search. In addition, so-PEGG handles a half-dozen 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 slight slowdown on two 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 a search trace serves only to *increase* runtime. The fact that so-PEGG suffers little relative to GP-e in this case is directly attributable to the low overhead associated with building so-PEGG's search trace. On the majority of problems that both me-EGBG and so-PEGG can solve, me-EGBG has the upper hand. Armed with a search trace designed to avoid most of the redundant consistency-checking effort, me-EGBG should be expected to dominate so-PEGG for problems with multiple search episodes, as long as memory constraints are not an issue. The fact that me-EGBG's advantage over so-PEGG is not greater for such problems, appears to be 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 overhead due to its more concise search trace. Note that there is no obvious advantage to prefer one state traversal order over the other *in non solution-bearing episodes* since both planners visit all the states in their PE for these search episodes.[9]

Now turning attention to the PEGG results, it's apparent that the beam search greatly extends the size of problem that can be handled. PEGG solves nine large 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. As indicated by the right-hand column of the table, PEGG solves prob-

[9] Empirically, we have in fact found advantages with respect to traversal order even in intermediate search episodes for some problems or domains. However, since this aspect is highly problem-dependent, we do not consider it in this study.

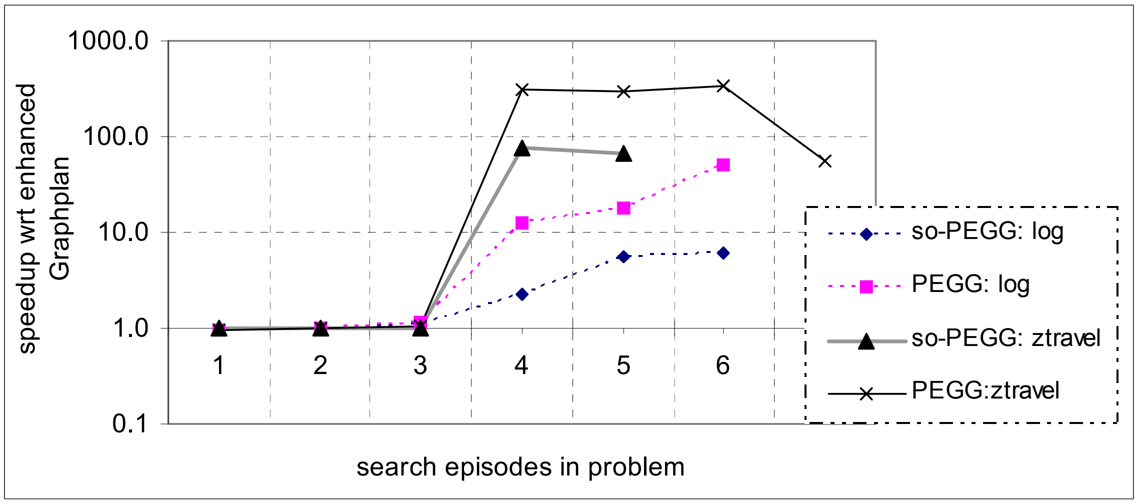

Figure 7. Speedup vs. number of search episodes: Logistics '00 and Ztravel '02 domains

lems up to 314 times faster than GP-e, the highly enhanced version of Graphplan. This is a conservative bound on PEGG's maximum advantage relative to GP-e since speedup values for the fifteen problems that GP-e fails to solve were conservatively assessed using the time limit of 1800 seconds.

Like EGBG, PEGG's use of a search trace is effective in problems with two characteristics; 1) Those for which the planning graph construction cost constitutes a modest fraction of overall runtime 2) Those requiring multiple search episodes prior to reaching a solution-bearing level. PEGG's strength with respect to the latter aspect is illustrated in Figure 7. 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 Ztravel domain from the AIPS-02 competition) and then averaging the speedups observed for problems with the same number of observed search episodes. Noting that the speedups are plotted on a logarithmic scale, we see the striking advantage of exploiting a search trace on multiple search episode problems. Moreover, 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.

5.2 The algorithm for the PEGG planners

Pseudo-code of the high level algorithm for so-PEGG and PEGG is given in Figure 8. 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 mutual 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 9 initialize the PE as they generate search segments holding the states generated during the regression search process. The former routine outlines the process of compiling 'conflict sets' (see Appendix B) as a means of implementing DDB and EBL during the action assignment search. The latter routine illustrates the role of the top level conflict set for recording a minimal no-good when

```
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*/
  {n, PG} ← find_1st_level_with_goals( Ops, Init, Goals )
  if n = Fail then Return FAIL /* graph reached level-off and goals are not present in non-mutex state */
  Goals ← order Goals according to variable ordering method
  SS_0 ←create search segment with fields:
                        goals←Goals, parent←'root, PE-level← 0, parent-actions← {}
  PE ← create pilot explanation structure with fields: ranked-segs ← {SS_0}, new-segs ←{}
  /* Conduct Graphplan-style backward search on the n-length planning graph, storing trace in PE..*/
  search-reslt ← assign_goals(Goals, {}, n, SS_0, 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 loop forever
      /* no n-length solution possible ...use the PE to search for a longer length solution */
      n ← n+1
      ranked-newsegs ← heuristic_sort( PE[new-segs] ) /* best first ranking of newly generated states */
      PE[ ranked-segs] ← merge sort heuristically ranked new segs into existing ordered PE segments list
      loop while there are unvisited search segments in PE[ranked-segs] (optionally: until all segments
            below the heuristic threshold have been visited)
         SS ← Select the highest ranked unvisited segment from PE[ranked-segs]
         k ← n – SS[PE-level] /* determine 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 loop.
         if SS[goals] ∈ memos(k, PG) /*compare against nogoods stored at level k */
           then remove SS from PE[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 Plan ← extract plan actions from the ancestors linked to search-reslt
                     Return Plan
               else search-reslt is a conflict set, C
                  Add C to memos(k, PG) /*memoize minimal nogood (form of EBL)*/
                  reorder SS[goals] ∴ goals∈ C appear first /*this is EBL-based reordering */
      end while
   end-loop
end
```

Figure 8. Top level algorithm for PEGG and so-PEGG planners

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 S_n (extending the PE) in *assign_next_level_goals* whenever S_n's goals are successfully assigned.

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 sorted and merged into the already ordered PE based on a state space heuristic ranking. An inner loop then considers each PE search segment in best-first order, and it is the logic here that primarily differentiates so-PEGG and PEGG. Whereas so-PEGG will visit every search segment

```
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,
              $SS_1$: search segment, PG: planning graph,  PE: pilot explanation (search trace)
ASSIGN_GOALS (G, A, k, $SS_1$, PG, PE)
 if G is empty or k = 0 (the initial state)
   Return $SS_1$  /* Success */
  else   /* there are goals left to satisfy…*/
    g $\leftarrow$ select goal from G
    cs $\leftarrow$ {g}   /*  initializes a conflict set for DDB */
    Ag $\leftarrow$ actions from PG level k that give g, ordered according to value-ordering heuristic
    forall  act$\in$ Ag
       search-reslt $\leftarrow$ {}
       if $\exists$ $\alpha\in$ A : action $\alpha$ is mutex with act
         then let b be the goal that $\alpha$ was assigned to support
              cs $\leftarrow$ cs $\cup$ {b}  /* add to conflict set and loop*/
        else   /* act has no conflict with actions already in A */
           if G is not empty
             then   /* search continues at this level with the next goal… */
                  search-reslt $\leftarrow$ assign_goals (G-{g}, A $\cup${act}, k, $SS_1$, PG, PE)
            else  /* no $SS_1$ goals left to satisfy...setup  for search at next lower level */
                  search-reslt $\leftarrow$ assign_next_level-goals (A $\cup${act}, k, $SS_1$, PG, PE)
           if  returned search-reslt is a conflict set, C
              if g$\in$ C then cs $\leftarrow$ cs $\cup$ C  /* absorb returned conflict set & try next action */
                else (g is not in the returned conflicts)   Return C
            else  returned search-reslt is a search segment,  Return search-reslt  /* ...Success */
    end-forall
    Return cs  /* no soln reached */
 end-if
 end

Set up search on graph level k-1 given that $SS_1$ goals have been satisfied by the actions in A at level k
ASSIGN_NEXT_LEVEL_GOALS (A, k, $SS_1$, PG, PE)
 nextgoals$\leftarrow$ regress $SS_1$[goals] over A, the actions assigned to satisfy goals
 if there is an M$\in$  memos(k-1, PG) such that M $\subseteq$nextgoals
   then Return M as the conflict set  /* backtrack: goals subsumed by nogood */
  else  /* ... initiate search on next lower PG level*/
     $SS_2$ $\leftarrow$ create new search segment with fields:
                goals$\leftarrow$ order nextgoals by variable ordering heur., parent$\leftarrow$$SS_1$,  parent-actions$\leftarrow$A
     Add $SS_2$ to PE[new-segs]
     search-reslt $\leftarrow$ assign_goals (nextgoals, {}, k-1, $SS_2$, PG, PE)
     if search-reslt is a search segment, Return search-reslt  /* Success */
       else search-reslt is a conflict set, C  /*memoize minimal nogood and return conflict set */
            add C to memos(k-1, PG)
            reorder $SS_2$[goals] $\therefore$  goals$\in$ C appear first  /*this is EBL-based reordering */
            Return search-reslt
 end
```

Figure 9. PEGG's regression search algorithm. Graphplan-style regression search on subgoals while concurrently building the search trace (PE)

whose goals are not found to match a memo, PEGG restricts visitation to a best subset, based on a user-specified criterion. Depending on the secondary heuristic used for ordering the PE segments[10], expansion of the planning graph is *deferred* until a segment is chosen for visitation that transposes to a planning graph level exceeding the current graph length. Consequently for some problems the PEGG planners may be able to extract a step-optimal solution while building one less level than other Graphplan-based planners.[11]

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 is 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, level-by-level. As for EGBG, search trace memory management is a challenge for PEGG once we stray from bottom-up traversal, but it is less daunting. We return to this after discussing 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 and 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 and 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 PEGG planners 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. One difference is rooted in the fact that the state space planner will visit a selected state only once while PEGG typically must consider whether to *revisit* a state in many consecutive search episodes. This is a consequence of the changing constraints associated with a state in the search trace as it is transposed to successive levels of the planning graph. 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

[10] The heuristic that accounts for flux (Section 5.3.2) reassesses this metric for a search segment based on the current level of the planning graph it transposes to. When this heuristic is employed, the decision to expand the planning graph is made during sorting as a search segment is chosen for heuristic re-evaluation.

[11] 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.

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, which will anyway 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 the potential for it to generate new local search branches.

In the next two subsections we discuss adaptation of known planning graph based heuristics for effective use with the search trace and then describe how they can be made more informed for selection of search trace states based on factors reflecting the potential for new search.

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

Since PEGG conducts regression search from the problem goals, 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 and Kambhampati, 2000). A 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 and Kambhampati 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 it 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 as the default heuristic for value ordering *as the graph is constructed*, 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 and Kambhampati 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 computed as 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 and 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.

The adjusted-sum heuristic contains a measure of the negative interactions between subgoals in a state, but not the positive interactions (i.e. 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 and 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.

5.3.2 Specializing a heuristic for the search trace

We describe here two specializations of the adjusted-sum heuristic to the search trace context. The first is a relatively straightforward form of heuristic updating that leverages PEGG's search experience to dynamically improve the h value estimate. 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 8 & 9). 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 achieves some of the flavor of a desired heuristic for ranking search trace states; the longer a state resides in the search trace, the more times it will have its h-value 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. The PEGG planners employ this update by default.

The second specialization focuses on boosting sensitivity to planning graph evolution as a search segment is transposed up successive levels. As discussed above, 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 visit *S* over other states in the trace 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* in the ranking of search trace states, so ideally the 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: appearance of a new action establishing 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. As such, h-values for states arising from any of the distance-based heuristics remain remarkably constant for any planning graph level beyond that at which the propositions appear and are binary non-mutex[12]. In contrast, we desire a measure that will 'improve' (i.e. reduce) a state's h-value in the event it is transposed to a planning graph level at which promising new branches of regression search open up. Conversely, when the graph levels off, search segments that are transposed into static levels typically have a lower potential for being the root of new search since, by definition, no new actions appear in levels at or above level-off.

We investigated a variety of hybrids of the distance-based heuristics that reflect 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.* This effort was guided by Observations A-1 and A-2 of Appendix A, which describe the three planning graph and memo cache properties that determine whether regression search on a subgoal set *S* will evolve over successive episodes (on successive planning graph levels):

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).

[12] This, in part, explains the observation (Nguyen and 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 the graph to level-off.

We will refer to these measures of the potential for new search branches to result from visiting a state in the PE 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 merit 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 requiring 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. Note however, that the graph mechanics underlying the relaxation of a dynamic mutex between a pair of actions at level *k* is 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 provide (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, 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 good estimate of flux for factors 1 and 2 above.

The third flux factor above is perhaps 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.*[13] So, ignoring the third factor, we have augmented the largely static adjusted-sum distance-based heuristic to reflect the two flux measures that depend on new actions. The resulting heuristic is sensitive to the evolution in search potential as a state is transposed to higher planning graph levels:

Adjsum-flux heuristic: *Define a state S associated with planning graph level k in search episode n:*

$$h_{adjsum-flux}(S) := \underbrace{\sum_{p_i \subset S} h(p_i) + (lev(S) - \max_{p_i \in S} lev(p_i))}_{\text{adjusted-sum terms}} \underset{\text{minus}}{-} \underbrace{w_1 * \frac{\sum_{p_i \subset S} newacts(p_i)}{|S|} - w_2 * \sum_{s_i \subset S_c} childflux(s_i)}_{\text{flux terms}} \tag{5-4}$$

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_1 and w_2 are weighting factors for the flux terms

Here the number of new actions establishing the subgoals of a state are normalized relative to the number of subgoals in the state. The *w1* and *w2* weighting factors bias the h-value towards either the

[13] 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. The no-good goals themselves contribute to the conflict set used to direct search within *S* whenever such backtracking occurs.

distance-based component or the measure of potential new search. Experimentally, over a broad range of problems, PEGG's performance improves relative to the adjusted-sum heuristic when the flux terms contribute between 0 and 30% of the h-value, but can degrade quickly for higher percentages . Optimal weighting values are highly problem-specific but *w1* = 1 and *w2* = .1 give reasonably robust performance over the domains we've studied, and are the default values for PEGG results in this report.

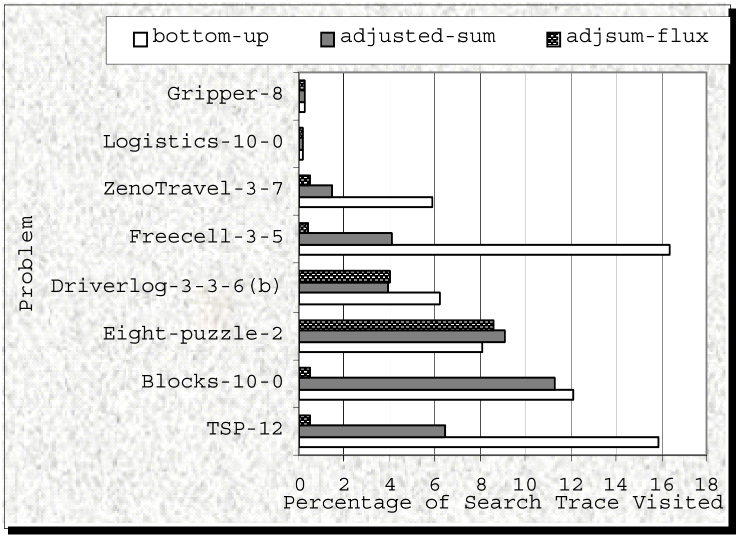

Figure 10: Comparison of accuracy for 3 heuristics: The metric is the number of PE search segments that must be visited in the final search episode before a solution is found.

Figure 10 provides evidence that the adjsum-flux heuristic outperforms both a bottom-up policy and adjusted-sum in so-PEGG. Here the heuristics are compared across a variety of problems in terms of the percentage of the total search trace segments that PEGG visits in the final episode prior to finding a seed segment (a state that can be extended to a solution). This is a direct measure of the power of the search segment selection heuristic. Performance can vary considerably with the specific problem but the charted examples are representative in our experience.

Three of the problems feature search traces on which the adjusted-sum does well and three are traces with seed segments that the adjusted-sum heuristic ranks too low (causing PEGG to visit them late). Clearly, adjsum-flux loses little, if any accuracy relative to adjusted-sum for the former problems and greatly improves on its accuracy in the latter.

PEGG's search history shows that the adjsum-flux heuristic also exhibits the desirable characteristic mentioned above regarding PE states that lie in levels above planning graph level-off. The planning graphs for all Figure 10 problems, except the Freecell problem, reach level-off prior to the final search episode. For problems such as Gripper-8, Eight-puzzle-2, Blocks-10-1, and TSP-12 the majority of the PE states in later search episodes lie in static levels. PEGG using the adjsum-flux heuristic not only ranks these search segments behind higher flux, non-static level states when appropriate, but has enough sensitivity to boost the rank of static level states when the only seed segment(s) reside in those levels. The former capability largely explains the advantage of adjsum-flux in Blocks-10-1 and TSP-12, while the latter property accounts for its dominance in Feecell-3-5.

5.4 Memory management under arbitrary search trace traversal order

Consider again the PE at the time of the final search episode in Figure 6. If we allow the search segments to be 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 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, not to mention the overhead associated with regenerating search segments. Less obvious is the heuristic information about a state that is, at least temporarily, lost when the state is regenerated instead of revisited as an extant PE search segment. Recall that for the adjusted-sum type 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. In addition, the adjsum-flux heuristic updates the flux value associated with each PE search segment prior to the search episode and this information is unavailable for the same state when newly generated.

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 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.[14]

Additional search trace memory management issues arise when we apply beam search on the PE. Under beam search PEGG will only visit 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 mini-episode 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. As such, we must retain as a minimum, the active search segments and all of their ancestor segments up to the root node. Beyond this requirement, there is a wide range of strategies that might be used in managing the inactive portion of the PE.

For this study we have not attempted to reduce PE memory requirements along these lines. We instead placed more emphasis on what might be termed the search space 'field of view'. The success of conducting beam search on the search trace depends significantly on how informed the heuristic is, and the states represented in the trace from which the heuristic selects. For a generally static heuristic estimate such as adjusted-sum, there is little incentive to retain a large portion of the inactive search segments in the PE since once they are determined to lie outside the heuristically best set there is little chance that they will ever move back into the active set. However, the adjsum-flux heuristic was specifically designed to dynamically respond to the evolution of the planning graph and the f-value of a given state can change significantly from one search episode to the next. This being the case, the larger the set of feasible states available to be compared heuristically, the more likely that a plan segment will be included in the active PE.

Since the much reduced memory footprint of PEGG's skeletal search trace makes it a much less critical issue, for this study we adopted a strategy of retaining in memory *all* search segments gener-

[14] In the interests of simplicity, the Figure 8 algorithm does not outline the memory management process described here.

ated. Their f-values are updated prior to ranking and selecting those to be visited in the beam search. The heuristic updating process is relatively inexpensive and this approach gives 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, where 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 from the planning graph, better to first attempt to assign values to the goals that are most difficult to assign.' 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 ASSIGN-GOALS routine used by PEGG to search on a search segment's 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 11 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.[15] Here we isolate the impact of EBL-based reordering from that of EBL itself in EGBG, by activating the EBL but using the conflict sets produced 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,

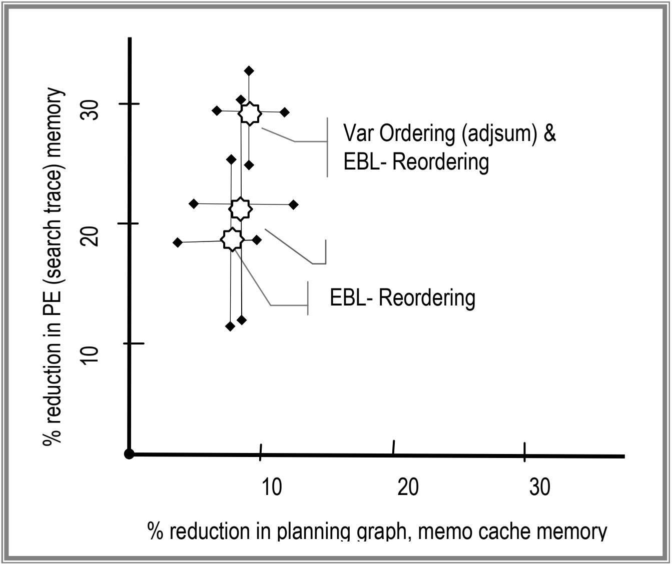

Figure 11: Memory demand impact along two dimensions for 'adjusted sum' variable ordering and EBL-based reordering techniques when applied independently and together (within EGBG).

[15] 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.

especially since unlike the adjusted sum, 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 11, 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. As such, it is not portable to any Graphplan-based planner we know of, including the GP-e system.

5.6 Trading off guaranteed step-optimality for speed and reach: Profile of PEGG under beam search

Many of the more difficult problems to solve with Graphplan's IDA* style search have 20 or more such search episodes before the reaching the episode in which a solution that can be extracted. The cumulative search time tied to these episodes can be a large portion of the total search time and, for many larger problems in Table 3, so-PEGG exhausts search time limits well before reaching the episode in which a solution can be extracted. PEGG abandons the search policy of Graphplan (as well as EGBG and so-PEGG) which exhausts the entire search space in each episode up to the solution bearing level. It is this very strategy that 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. In this study we focus on 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, as so-PEGG does, 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.

Returning to 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. The intent is to avoid visiting (and optionally, not even retaining) search segments that hold little promise of being extended into a solution, as predicted by the secondary heuristic used. When the first segment exceeding this threshold is reached on the sorted queue the search episode ends.

Devising 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.

There are a wide variety of strategies for conducting beam search on PEGG's search trace. The next two subsections touch briefly on two important influences in shaping the approach we adopted.

5.6.1 Using flux to filter the beam

As an adjunct to distance-based heuristics, the flux measure detailed in section 5.1.2 provides a modest improvement in search performance for the final search episode. Under beam search the flux measure can much more strongly impact every search episode as it influences the states actually included in the active PE. Moreover, empirically we find that the same flux measure exhibits even greater impact as a filter. 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. The intuition here is that a state which qualifies for the active PE due to a low f-value based entirely on the (largely static) distance heuristic, is not likely to be worth visiting if its flux does not exceed some minimal threshold. That is, it is only worthwhile to (re)visit a search segment at the new planning graph level it is associated with *if* it has some potential for engendering new search branches *above and beyond* any its descendents might produce. The flux metric introduced in section 5.1.2 and already used in the adjsum-flux heuristic estimates precisely this.

This flux measure actually proves more effective as a filter than it does as an adjunct to the secondary heuristic PEGG uses to direct traversal of its search space. This is due, in part, to the difficulty of finding values for the weights of the adjsum-flux heuristic (eqn 5-4) that are highly effective across a wide variety of domains and problems. The size of the flux terms in equation 5-4 can vary by factors of 10 - 100 across problem domains and for a given set of weights may either overwhelm the adjusted-sum terms or become insignificant. However, when the flux terms are used as a filtering measure against a threshold of '0' (which we find to be the most effective value), the weights become irrelevant. Since the flux metric is indeed just an estimate of the potential for search on a state to generate new search branches, it's possible for such new search to occur under a search segment with 0 flux. Empirically the estimate proves accurate enough to often filter out hundreds of states that are otherwise candidates to be visited without pruning plan segments in the step-optimal solution.

5.6.2 PEGG's plan quality profile

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 will tend to generate non-optimal plans more quickly, often containing redundant or unnecessary actions. As alluded to in Section 5.1, where the settings adopted for the Table 3 results are discussed, those PEGG experiments are biased towards producing optimal makespan solutions.

The step-optimal plan produced by enhanced Graphplan is matched by PEGG for all but four of the 36 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]. (Problems for which PEGG returns a solution with longer makespan than the step-optimal have boldface step/action values.) In these four problems, PEGG returns solutions within two steps of optimum, in spite of the highly pruned search it conducts. We found this to be a fairly robust property of PEGG's beam search under these settings across all problems tested to date.

Problem	**'N-best first'** [state-space search, N=100	**PEGG** adjsum-flux heuristic, beam search on100 best search segments	**SATPLAN** (optimal)
bw-large-a	8	6	6
bw-large-b	12	9	9
bw-large-c	21	14	14
bw-large-d	25	18	18

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

PEGG often finds plans with fewer *actions* than GP-e in parallel domains and it is interesting to note that this 'Graphplan hybrid' system is also impressive in serial domains such as blocksworld (which are not exactly Graphplan's forte).

A variety of methods of trading off the guarantee of finding an optimal length plan in favor of reduced search effort have been investigated in the planning community. By comparison, PEGG's beam search approach appears to be 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 (Approximating the N-best algorithm, PEGG is run with 100 search segments 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 we focus only the length of the plans produced here. 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 12 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. LPG's results are

[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.

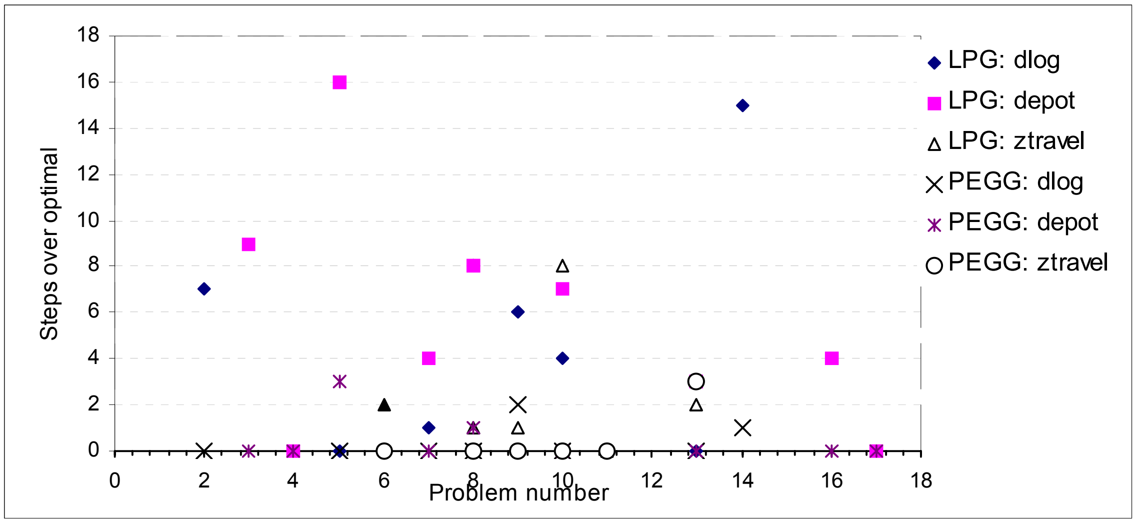

Figure 12: Makespan comparison of PEGG and LPG -Departure from step-optimal plan length. (LPG data taken from the AIPS 2002 competition results.)

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 PEGGs. LPG, too, can be biased to produce plans of higher quality (generally at the expense of speed) and here we report its performance when operating in the 'quality' mode. PEGG's exploitation of the search trace excels in this regard, as its maximum deviation from optimum is three steps, and most of the plot points for its solutions lie right on the optimal makespan axis. In terms of number of actions instead of steps, LPG and PEGG results on these problems tend to be more comparable. One explanation may be that there are sets of actions within LPG's solutions that could be conducted in parallel.

We chose these particular problems because the optimal solution is known, and we are interested in comparing to a quality baseline. 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 Comparison to heuristic state space search

We do not yet have an equitable platform for comparing PEGG's runtimes against other planners from the most recent planning competition that are capable of producing parallel plans. The version of PEGG reported here is written in Lisp and all experiments were run on a 900 Mhz laptop with 384 MB RAM, while the competition planners were generally coded in C and the published results are based on the execution on the competition machines. Given PEGG's close coupling with the planning graph, the most relevant comparisons might be made 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 one of the fastest distance-based heuristic state space planners using most of the same problems as Table 3. AltAlt (Srivastava, Nguyen, 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 for two of the most effective heuristics developed for that planner (Nguyen and Kambhampati, 2000), the first of which is the version of the adjusted-sum heuristic described in section 5.3.1. 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 very recent work with a highly modified version of AltAlt does, in fact, construct such plans -Nigenda and 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

Perhaps the feature that most distinguishes the EGBG, me-EGBG, so-PEGG, and PEGG systems from 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. In addition, they each confront in different ways a com-

Problem	**PEGG** heuristic: adjsum-flux cpu sec (steps/acts)	**Alt Alt** (Lisp version) cpu sec (/ acts) heuristics:	
		adjusum2	combo
bw-large-B	4.9 (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 (/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)	20.92 (/59)
Tower-7	6.1 (127/127)	7.0 (/127)	*
Tower-9	23.6 (511/511)	28 (/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	8.9 (12/12)	21.1 (/12)	18.9 (/12)
AIPS 1998	PEGG	Alt Alt (Lisp version)	
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 (/53)
log-y-5	30.5 (16/34)	44 (/38)	29 (/42)
mprime-1	2.1 (4/6)	722.6 (/ 4)	79.6 (/ 4)
AIPS 2000	PEGG	Alt Alt (Lisp version)	
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	58.7 (54/54)	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	52.9 (6/10)	49 (/12)	
schedule-8-9	155 (5/12)	123 (/15)	
AIPS 2002	PEGG	Alt Alt (Lisp version)	
depot-6512	2.1 (14/31)	1.2 (/33)	
depot-1212	127 (22/56)	290 (/61)	
driverlog-2-3-6e	66.6 (12/26)	50.9 (/32)	
driverlog-4-4-8	889 (23/38)	461 (/44)	
roverprob1423	15 (9/28)	2.0 (/33)	
roverprob4135	379 (12 / 43)	292 (/48)	
roverprob8271	444 (11 / 39)	300 (/ 45)	
ztravel-3-7a	222 (11/24)	77 (/28)	
ztravel-3-8a	3.1 (9/26)	15.4 (/31)	

Table 5. PEGG and a state space planner using the 'adjusted-sum heuristic

PEGG: bounded PE search, best 50 search segments visited in each search episode, as ordered by adjsum-flux state space heuristic

AltAlt: Lisp version, state space planner using planning graph distance-based heuristic. "adjusum2" and "combo" are the most effective heuristics

Allegro Lisp platform, runtimes (excl. gc time) on a Pentium III, 900 mhz, 384 M RAM mon problem faced by learning systems; the relative utility of the learned information versus the cost of storing and accessing the information 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 the 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 the PE in search episode e

Equation 6-1 indicates that amortization of the high overhead of logging consistency-checking experience during search on subgoal sets (search segments) 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 in fact, 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).

We adopted a straightforward approach to capturing and reusing Graphplan's consistency-checking effort in a search episode, essentially logging the results of mutex checks in a bit vector and then replaying them to minimize redundant checking in the next episode (see Appendix A). This has its drawbacks, as evidenced by the sixteen problems of Table 3 for which even the memory-efficient version of EGBG encounters memory limitations. In addition, the tightly ordered relationships between the action assignment bit vectors makes it difficult to fully accommodate what proves to be an important adjunct to learning for these systems; dependency directed backtracking. The full benefit of DDB is sacrificed when the dictates of the action assignment vectors are adhered to.

The PEGG results of Tables 3 and 5 demonstrate that the utility of a search trace is not limited to minimizing redundant consistency-checking effort. 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 candidate set of partial plans with each segment's state being the current tail state of such a plan. Our experimental 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 can be accomplished with greatly reduced memory demand. Freed from the need to regenerate the RS reachable states in IDA* fashion during each search episode, PEGG can directly revisit any such state in an attempt to extend the partial plan to a solution, if it appears sufficiently promising.

There is an important aspect in which PEGG's beam search on the PE states differs from an 'N-best first' approach for a state space planner, such as reported in Table 4. When conducting search, 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. We could instead adopt a greedy approach by also applying the heuristic bound during this regression search. That is, we could backtrack whenever a state is generated that exceeds the f-value bound restricting which search segments are visited. This can be seen as a translation of the Greedy Best First Search (GBFS) algorithm employed by HSP-r (Bonet and Geffner, 1999) for state space search, into a form of hill-climbing search on the planning graph.

Experimentally we found that, when PEGG was adapted to enforce the PE state f-value limit during its regression search, improvements were unpredictable at best. Speedups of up to a factor of 100 were observed in a few cases (all logistics problems) but in most 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 primary factors that appear to explain this result: 1) PEGG's regression search is greatly expedited by EBL as well as DDB, but the regressed conflict set that these techniques rely on is undefined when the f-value limit precludes search on a child state. Without conducting such search there is no informed basis for returning anything other than the full set of subgoals in the state, and returning the entire state generally undermines the EBL/DDB process for continued search on the ancestors of the child state. 2) It's difficult to assess an f-value for a newly generated state to compare against an f-value bound that is based on PE states generated in previous episodes. The heuristic values of PE states that determine the f-value bound have been both *increased* based on PEGG's use of search experience to improve h-value estimates each time a state is visited (Section 5.1.2), and *decreased* due to the flux adjunct in the adjsum-flux heuristic. When PEGG operates with the f-value bound enforced in its regression search the second factor can cause it to occasionally prune a plan segment that would otherwise have been visited.

The degradation of solution quality as we shift PEGG closer to a greedy search approach such as this, may be an indicator that PEGG's remarkable 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

In terms of 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. We will *not* discuss further the wide assortment of work directly related to some of the search techniques, efficiencies, and data structures that underpin the ability of EGBG and PEGG to successfully employ a search trace. We endeavored to cite relevant sources above wherever such ancillary topics first arose: These include investigations of more memory efficient bi-partite representations of the planning graph, explanation based learning and dependency directed backtracking in the context of search on the planning graph, variable and value ordering strategies, and the evolution of distance-based heuristics, as well as the potential for extracting them from the planning graph.

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 can be seen as directly addressing this shortcoming by serving as a memory of the *states* in the visited search space of the previous episode in order to reduce redundant regeneration. In this respect PEGG's search is closely related to methods such as MREC (Sen, Anup and 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. Noteworthy differences include the backing up of a deleted node's f-value to the parent node in all three of the above algorithms. 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 memory limitations due to PE size.

EGBG and PEGG are the first planners to directly interleave the CSP and state space views in problem search, but interest in synthesizing different views of the planning problem has lead to some related approaches. The Blackbox system (Kautz and 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 and Kambhampati, 2000), similarly alternates between extending a planning graph and converting it, but they transform the structure into CSP format and search to satisfy the constraint set in each search phase.

The beam search concept is employed in the context of prepositional satisfiability in GSAT (Selman, Levesque, Mitchell, 1992) and is an option for the Blackbox planner (Kautz and 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 a number of 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 and 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 considering 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. For example, 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 planning 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 prohibited in terms of memory requirements if we had to store multiple versions of the PE. However we can retain only the one version of it and simply store any level-specific heuristic information in its search segments as values indexed to their associated planning graph levels. More challenging issues include such things as what range of plan lengths should be considered at one time and how to avoid having to deal with plans with steps consisting entirely of 'persists' actions.

We haven't specifically 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* (Learning Real-Time A*, Korf, 1990) and variants such as B-LRTA* (Bonet, Loerincs, and 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 may be simultaneously transposed onto multiple planning graph levels. In this mode PEGG would be biased to search quickly for a plan of any length, and subsequent to finding one, could 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 and 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 returns 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 and me-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 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, sacrificing EGBG's ability to minimize redundant consistency checking during search in favor of a design more conducive to informed traversal of the search trace states. 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 extend them with a metric we call "flux", which is targeted at planning graph search. 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. However its advantage is apparent in the greatly extended range of problems it can handle, exceeding available memory for only one problem of our test set, versus 16 failures for me-EGBG. More dramatic evidence of the speedup potential for a search trace guided planner is provided by PEGG, which employs beam search based on the heuristic values of states in its search trace. 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 when we limit the beam search to just 50 states in each episode, PEGG returns the step-optimal plan in almost 90% of the test bed problems and comes within one or two steps of optimal in the others. It does so at speedups ranging over two orders of magnitude above GP-e, and quite competitively with a state-of-the-art state space planner (which finds only serial plans).

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

References

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

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

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

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

Fox, M., Long, D. (1998). The automatic inference of state invariants in TIM. JAIR, 9:317-371.

Frost, D. and 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. JAIR 5:95-137.

Gerevini, A. Serina, I., (2002). LPG: A planner based on local search for planning graphs with action costs. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling.* Toulouse, France, April, 2002.

Haslum, P., Geffner, H. (2000). Admissible Heuristics for Optimal Planning. In *Proc. of The Fifth International Conference on Artificial Intelligence Planning and Scheduling.*

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. Vol 105.*

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, 2000.

Kambhampati, S. and 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 4th European Conference on Planning.

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

Kautz, H. and 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, pages 273-285, Springer LNAI 1348.

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. and Fox, M. (1999). Efficient implementation of the plan graph in STAN. *JAIR*, 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. *To appear: JAIR 2003.*

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

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; 10th European Conference on Artificial Intelligence, pp 1-5, Vienna, Austria.

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. University of Washington.

Srivastava, B., Nguyen, X., Kambhampati, S., Do, M., Nambiar, U. Nie, Z., Nigenda, R., Zimmerman, T. (2001). AltAlt: Comjbining Graphplan and Heuristic State Search. In *AI Magazine*, 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. and 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. Sixth International Conference on Artificial Intelligence Planning and Scheduling, workshop on Planning and Scheduling with Multiple Criteria, Toulouse, France. April, 2002.

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

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 observations that are entailed:

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. Any mutexes between pairs of actions that are establishers of propositions of S at level k are still 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 at the time it was generated, 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 observation A-1 in mind, we wish to exploit the search trace in a new episode to conduct sound and complete search while limiting search effort to only three situations: 1) under state variables with newly extended value ranges (i.e. search segment goals that have at least one new establishing action at the new associated graph level) 2) at points in the previous search that backtracked due to attempts to assign values that violate a 'dynamic' constraint (i.e. two actions that were dynamic mutex) and 3) checking the states that matched a cached memo in episode *n* against the memo cache at the next higher level in episode *n+1.* All other assignment and mutex checking operations involved in satisfying the goals of *S* 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 to satisfy 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 goals 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.

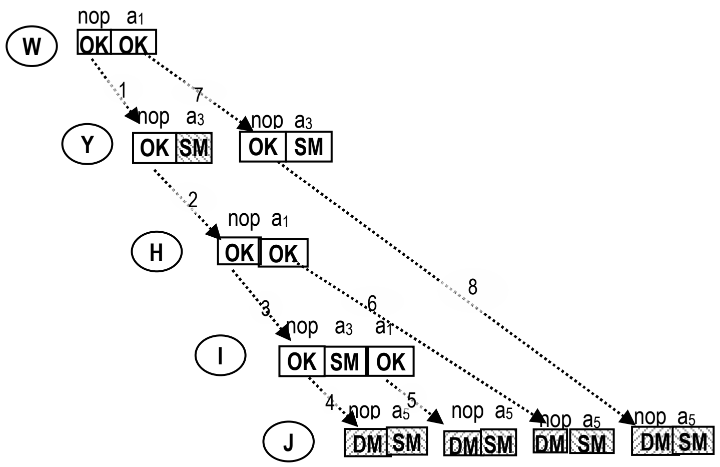

Figure A1. Bit vector representation of the search trace for the WYHIJ state in Figure 3. [Semantics: OK> assigned (no action conflicts) SM> action is static mutex with previous assign DM> action is dynamic mutex with previous assign NG> a no-conflict action results in a no-good state at lower graph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

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 mutual mutex conditions. (This is the call to *find_1st_level_with_goals*). The first search episode is conducted in the same fashion as for Graphplan, 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 segment is captured in trace vectors as described above. These search segments are stored in the PE structure, indexed according to the level at which they where generated (with 0 corresponding to the problem goals state at the current highest planning graph level).

Subsequent to the first episode, *EGBG_plan* enters an outer loop that employs the PE to conduct successive episodes. The search strategy alternates between the selection and visitation of a promising state from the search 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 is only called when a state being visited lies 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 capturing 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 comes up for assignment, 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.

```
 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,
              SS_1: search segment, PG: planning graph,  PE: pilot explanation (search trace)
ASSIGN_GOALS (G, A, k, SS_1, PG, PE)
  g-assigns ← {}   /* trace vector will hold ordered action assignment tags */
  if G is empty or k=0 (the initial state)
    Return SS_1  /* Success */
   else   /* there are SS_1 goals left to satisfy…*/
     g ← select goal from G
     Ag ← set of actions from PG level k that give g
     forall  act ∈ Ag
        search-reslt ← {}
        if ∃ α ∈ A : α is dynamic mutex with act then g-assigns ← append 'dm' tag
         else if ∃ α ∈ A : α is static mutex with act then g-assigns ← append 'sm' tag
         else   /* act has no conflict with actions already in A */
            if G is not empty
             then  /* search continues at this level with the next goal… */
                g-assigns ← append 'ok' tag
                search-reslt ← assign_goals (G-{g}, A ∪{act}, k, SS_1, PG, PE)
             else  /* setup  for search at next lower level */
                search-reslt ← assign_next_level_goals (A ∪{act}, k, SS_1, PG, PE)
                if search-reslt = 'nogood' then g-assigns ← append 'ng'
                 else g-assigns ← append 'ok' tag   /*  search occurred at lower level */
            end-if
        end-if
        if search-reslt is a search segment
          then  Return search-reslt  /* Success */
         else loop /*  try another action…*/
     end-forall
     push g-assigns → SS_1[trace]  /*add trace data to search segment */
     Return nil  /* no soln reached */
  end-if
end

Set up Graphplan-style search on graph level k-1 given that SS_1 goals have been satisfied by the
actions in A at level k
ASSIGN_NEXT_LEVEL_GOALS (A, k, SS_1, PG, PE)
  nextgoals ← regress SS_1[goals] over A actions
  if nextgoals ∈ memos(k-1, PG) then Return 'nogood'  /* backtrack due to nogood goals */
   else  /* ... initiate search on next lower PG level*/
      SS_2 ← create new search segment with fields:
                    goals ← nextgoals, parent ← SS_1, parent-actions ← A
      Add SS_2 to PE[max PG level – (k-1)]
      search-reslt ← assign_goals (nextgoals, {}, k-1, SS_2, PG, PE)
      Add nextgoals to memos(k-1, PG)  /*memoize nogood */
      Return search-reslt
 end
```

Figure A3.   EGBG's open regression search algorithm

```
Replaying regression search captured in search trace (PE) to focus effort on only new search possibilities
REPLAY_TRACE_GOALS  (G, A, k, $SS_{assigns}$ ,$SS_1$, PG,PE)
  if G is empty /* all $SS_1$ goals in this branch were successfully assigned during  last episode...*/
    then Return  /* .. just continue with level k replay, ignoring search replay at next lower level  *
   else  /* there are goals left to satisfy…*/
      g ← select goal from G;    g-assigns ← pop front trace vector from $SS_{assigns}$
      Ag ←set of actions from PG level k that give g
      /* Now replay assignments for g  from previous episode, rechecking only those that may have changed..*/
      forall  tag∈  g-assigns
          search-reslt ←{}
          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, $SS_{assigns}$, $SS_1$, PG, PE)
            else if  tag = 'sm' then loop  /* act was static mutex with action in A last episode –will persist*/
            else if  tag = 'dm'  /* act was dynamic mutex with action  in A last episode …retest..*/
                if mutex persists then loop
                  else change tag to 'ok' in g-assigns vector  /* act is no longer mutex with A actions */
                    if G is not empty  /*  resume backward search at this level… */
                       then search-reslt ← assign_goals (G-{g}, A ∪{act}, k, $SS_1$, PG,PE)
                     else  /* no goals left to satisfy in $SS_1$, setup for search at lower level */
                       search-reslt ← assign_next_level_goals (A ∪{act}, k, $SS_1$, PG, PE)
                       if search-reslt ='nogood' then change g-assigns vector tag to 'ng'
            else-if  tag = 'ng' then loop   /*...no conflict in last episode, but regressed subgoals are nogood */
          if search-reslt is a search segment  then  Return search-reslt  (Success) else loop (check next action)
      end-forall  /* 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, $SS_1$, PG, PE)
             if search-reslt is a search segment  then  Return search-reslt  /* Success */
       else push g-assigns → $SS_1$[trace]
      Return nil  /* no solution found in search stemming from $SS_1$ goals */
  end-if
end

Attempt to assign new actions that establish goal g at level k while building the search trace
ASSIGN_NEW_ACTIONS (G, A, Ag, g, g-assigns, k, SS, PG, PE)
    forall nact∈  Ag     (i.e. new actions establishing g)
         search-reslt ←{}
         if ∃ α∈  A : α is dynamic mutex with nact then g-assigns ← append 'dm' tag
          else if ∃ α∈  A : α is static mutex with nact then g-assigns ← append 'sm' tag
          else    /* nact has no conflict with actions already in A */
             if G is not empty
               then   /* new search resumes at this level with the next goal… */
                     g-assigns ← append 'ok' tag
                     search-reslt ← assign_goals (G-{g}, A∪{nact}, k, SS, PG, PE)
               else  /* setup  for new search at next lower level */
                     search-reslt ← assign_next_level_goals (A∪{nact}, k, SS, PG, PE)
                     if search-reslt = 'nogood' then g-assigns ← append 'ng'   /* ..subgoals are  nogood */
                      else g-assigns ← append 'l-ok' tag   /*  search occurred at lower level */
             if search-reslt is a search segment  then  Return search-reslt  (Success) else loop
    end-forall
  push g-assigns → SS[trace]
end
```

Figure A4.   EGBG's search trace replay algorithm

Appendix B: Exploiting CSP speedup methods to reduce memory demands

Background and implementation details are provided here for the six existing 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 and Long, 1998, Gerevini and 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. Domain preprocessing for me-EGBG /PEGG is basic relative to the above work, 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. This is done prior to the start of planning. The architecture discriminates between static (or permanent) mutex relations and dynamic mutex relations (in which a mutex condition may eventually relax) between actions and proposition pairs and use this information 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 a benchmark '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. It has been known for some time that this multi-level planning graph could be efficiently represented as an indexed two-part structure (Fox and Long 1998, Smith and Weld, 1998). Finite differencing techniques can then be used to focus on only those aspects of the graph structure that can possibly change as it is incrementally extended, leading to more rapid construction of a more concise planning graph.

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. In order to access the related proposition at the next higher graph level in a subsequent search episode, a search for the term in the proposition level must be conducted. Of course, the planning graph could be modified by adding pointers connecting related propositions/actions at added memory cost, but this makes more awkward the kind of rapid access of the constraint profile for these structures that we will ultimately find useful in versions of the PEGG planner.

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 and Kambhampati, 1999), where the primary interest was in their speedup benefits. Although the techniques were shown to result in modest speedups on several small problems, 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 will 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.

In the manner of the conflict directed back-jumping algorithm (Prosser, 1993), the failure explanations are compactly represented in terms of "conflict sets" that identify the specific action/goal assignments that have resulted in backtracking. This liberates the search from chronological backtracking, allowing it to jump back to the most recent variable taking part in the conflict set. In addition, when all attempts to satisfy a set of subgoals (state) fail, the complete conflict set that is regressed back represents a valuable '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 has smaller memo caches in terms of memory.

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 EGBG and PEGG have been outfitted with EBL/DDB for all non-PE directed Graphplan-style search. 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.[17] 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 use of memory by EGBG/PEGG to build and maintain 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 only has to be computed once.[18] 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.

[17] Briefly, the set level for an action, proposition or proposition set is the first level in which it appears in the planning graph. The adjusted-sum heuristic for a set of propositions adds to set level the difference in levels between the first planning graph level at which the set are all present and the level at which they become pair-wise non-mutex.

[18] Although neither the original Graphplan nor the ordering augmented version in Kambhampati, 2000 actually orders their actions during graph construction, in principle, this could be done for any planner that builds a planning graph.