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 strategy that employs available memory to construct a search trace, which is used to transform the depth-first, IDA* nature of Graphplan's search into an iterative state space view. We present a family of methods, each of which exploits a variant of the search trace to learn from different aspects of Graphplan's iterative search episodes in order to expedite search in subsequent episodes. The *EGBG* planners successfully avoid much of Graphplan's redundant search effort, while the *PEGG* planners 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. We demonstrate that distance-based heuristics can be adapted to informed traversal of the search trace and develop an augmentation of these heuristics targeted specifically at planning graph search. Guided by such a heuristic, the step-optimal version of *PEGG* 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.

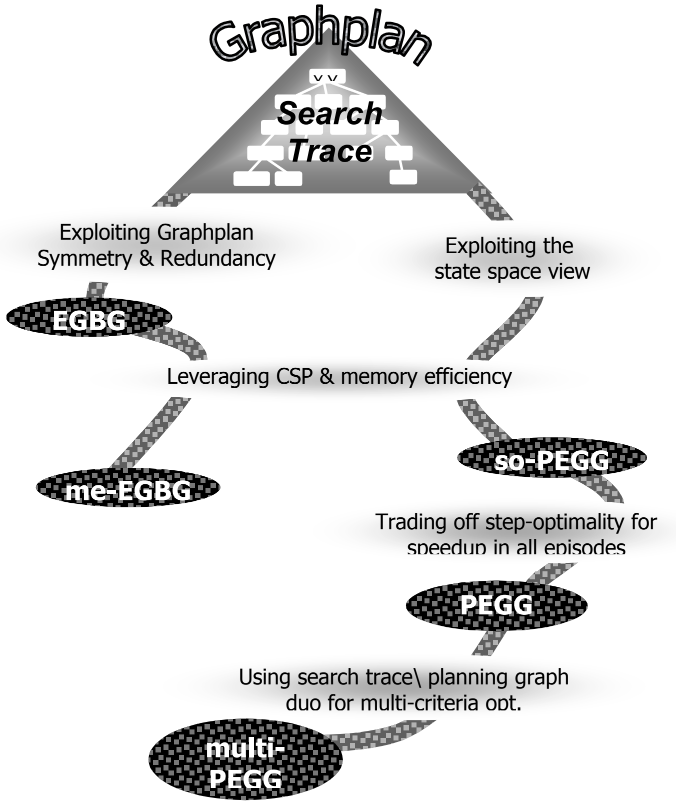

Graphplan
Search Trace
Exploiting Graphplan Symmetry & Redundancy
Exploiting the state space view
EGBG
Leveraging CSP & memory efficiency
me-EGBG
so-PEGG
Trading off step-optimality for speedup in all episodes
PEGG
Using search trace\ planning graph duo for multi-criteria opt.
multi-PEGG

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 the search trace approach to expediting search on the planning graph, we explored several variations that differed 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 employ a more comprehensive (and memory intensive) trace focused on minimizing redundant search effort.
- *right track:* The PEGG planners 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 a variety of 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 (implemented as the me-EGBG system).

The attention to memory efficiency markedly improves the speed and capabilities of the planner, but still leaves us with a variety of problems that lie beyond the planner's reach due to memory constraints. This motivates a shift to a greatly pared down search trace that forfeits minimization of redundant search in exchange for a much smaller memory footprint and an enhanced view of the search space as a set of states that can be visited in a more informed order. The strategy not only reduces memory overhead, but also greatly reduces the computational time spent building and revising the search trace over consecutive episodes. This third approach we describe and implement 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 ability to overlay a 'secondary heuristic' on top of Graphplan's (implicit) admissible IDA* heuristic, thereby allowing it to visit the search space encapsulated in the search trace in a more intelligent order. We examine the 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) to the task of traversing the search trace and then examine their shortcomings in this regard. This leads us to develop a specialized measure of the potential for a state in the trace to seed new search branches beyond those that were generated (and failed) in previous search episodes. We demonstrate how this 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, the PEGG-track planner 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 to expediting planning graph search with a search trace that 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 employing a search trace; 1) further reduction in search trace memory demands 2) effective release from Graphplan's exhaustive search of the planning graph in *all* search episodes. Our experimental results indicate 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 make-span, is virtually at the optimal.

The last planner in the PEGG track of Figure 1, multi-PEGG, is included for completeness but is not reported on in this paper. Multi-PEGG (Zimmerman and Kambhampati, 2002), exploits the unique advantage of the combined planning graph and search trace structures to address multiple plan quality criteria.

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 embodies a great deal of information that defines and constrains the search space traversed by Graphplan-style search, and the approaches we investigate here heavily exploit this structure to minimize the additional information needed for an effective search trace. 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.

An interesting upshot of employing available memory to build and maintain a search trace is that it allows us to adopt 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'. The search trace enables PEGG, in particular, to adopt a global view of the regression search state-space generated in episode *n*, and to select particular regions of that state-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 is also able to learn in an entirely 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.

We will show that employing available memory to construct and maintain a concise search trace, affords planning graph-based planners several unique capabilities. The sound and complete so-

PEGG planner often returns optimal plans with *parallel* actions one or more orders of magnitude faster than even a highly enhanced version of Graphplan. The PEGG planner trades off the optimality guarantee (and indeed, completeness, since it employs beam search) so as to boost its performance above a state-of-the-art heuristic forward state space planner. In this mode, PEGG generally dominates on planning problems in both serial and parallel domains, in terms of speed and still manages to return the step optimal plan in the great majority of cases.

We organize the paper 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 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. Section 4 concerns our investigations into a variety of memory reduction techniques and reports the impact of a combination of six of them on the performance of EGBG. The evolutions that led to 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 compares this work to related research. Section 8 gives our conclusions.

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

In this section, 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) a 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 (see Figure 2). We start with the initial state as the zeroth level proposition list. Given a k-level planning graph, the extension of the graph structure to level k+1 involves introducing all actions whose preconditions are present in the k^{th} level proposition list. In addition to the actions given in the domain model, we consider a set of dummy ``persist'' actions, one for each condition in the $k^{th}$ level proposition list. A ``persist-C'' action has C as its precondition and C as its effect. Once the actions are introduced, the proposition list at level k+1 is constructed as just the union of the effects of all the introduced actions. The planning graph maintains the dependency links between the actions at level *k+1* and their preconditions in level k proposition list and their effects in level *k+1* proposition list.

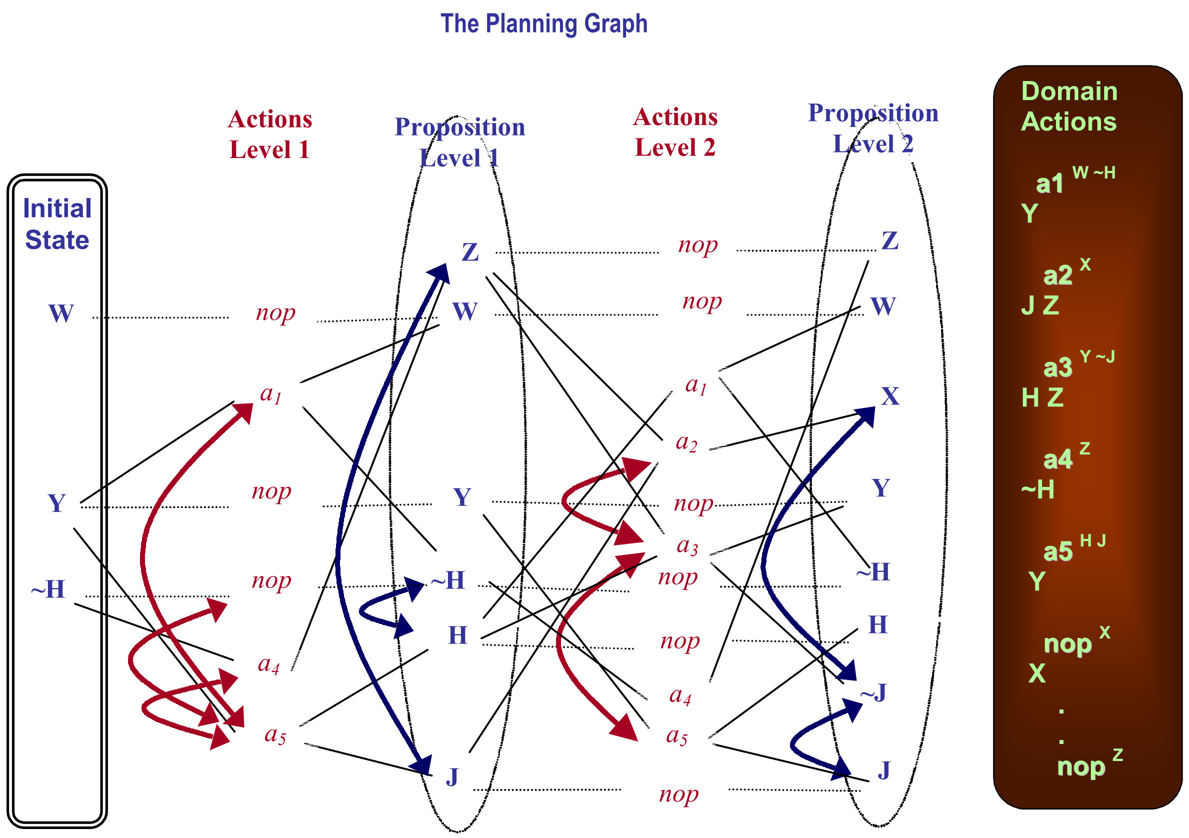


Figure 2. *Portion of a planning graph for an example domain*

Action descriptions: $_{[preconditions]}\, action_{\#}{}^{[effects]}$

The planning graph construction also involves computation and propagation of binary "mutex" constraints. The propagation starts at level 1 by labeling as mutex, all pairs of actions that are statically interfering with each other (i.e., their preconditions and effects are logically inconsistent). Mutexes are then propagated from this level forward using two simple propagation rules. In Figure 1, the curved lines with x-marks denote the mutex relations: two propositions at level k are marked mutex if all actions at level k that support one proposition are mutex with all actions that support the second proposition. Two actions at level k+1 are mutex if they are statically interfering (“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 (termed “dynamic mutex” since this constraint may relax at a higher planning graph level). 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).

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 goals at level k, we select an action from the level k action list that supports it, such that no two actions selected for supporting two different goals are mutually exclusive (if they are, we backtrack and try to change the selection of actions). 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. 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). This process can be viewed as a system for solving "Dynamic CSPs" (DCSP) (Kambhampati 2000, Hoffman, 2001,), wherein the standard CSP formulism is augmented with the concept of variables that do not appear (a.k.a. get activated) until other variables are assigned.

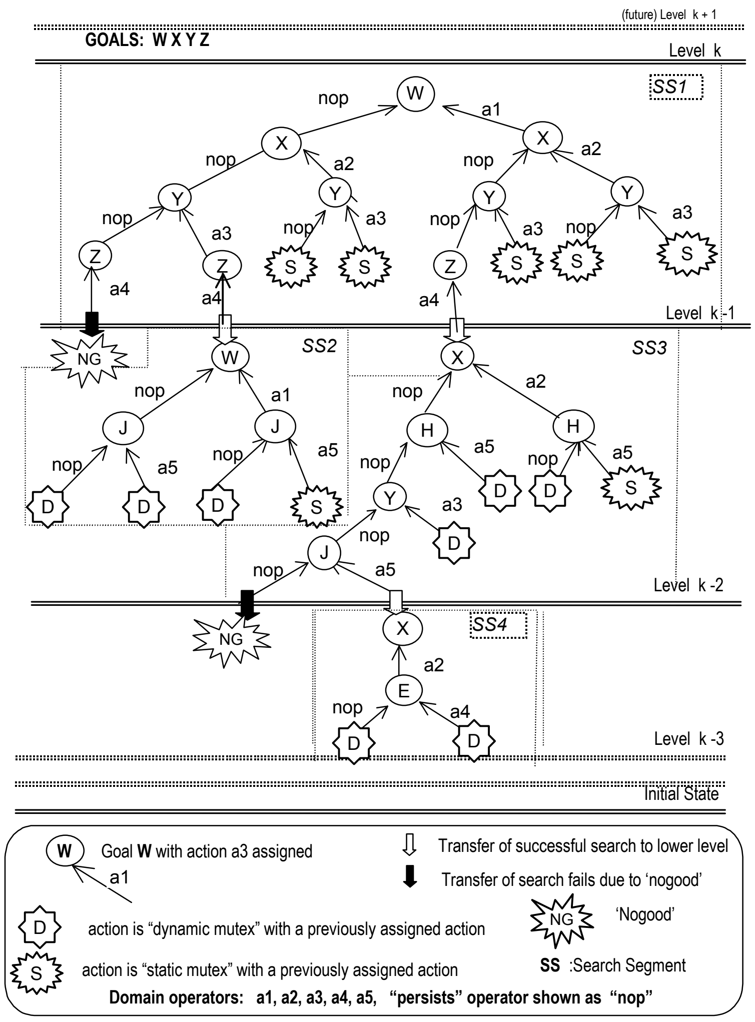

(future) Level k + 1
GOALS: W X Y Z
Level k
SS1
nop
a1
a2
a3
a4
a5
W
X
Y
Z
J
H
E
S
D
NG
Level k -1
SS2
SS3
Level k -2
SS4
Level k -3
Initial State
Goal W with action a3 assigned
Transfer of successful search to lower level
Transfer of search fails due to 'nogood'
action is "dynamic mutex" with a previously assigned action
'Nogood'
action is "static mutex" with a previously assigned action
SS :Search Segment
Domain operators: a1, a2, a3, a4, a5, "persists" operator shown as "nop"

Figure 3. *CSP-style trace of backward search at level k of a (vertically oriented) planning graph*

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. A sufficient condition defining this state is a level where no new actions are introduced *and* no existing propositions have a newly relaxed dynamic mutex. We call this 'level-off' and 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 an elemental form of nogood learning. When a set of (sub)goals for a level k is determined to be unsolvable, they are *memoized* at that level in a hash table. Correspondingly, when the backward search process later enters level k with a set of subgoals, they are first checked against the hash table to see if they have already been proved unsolvable. The legend in Figure 3 explains the figure icons that depict the different conditions that can result in backtracking, static mutex actions, dynamic mutex actions, and no-good states that have been memoized.

In the next subsection, we discuss Graphplan's search process from a higher-level perspective that abstracts away its CSP nature. Before doing so, we note that like other types of CSP-based algorithms, Graphplan consumes most of its computational effort on a given problem in checking

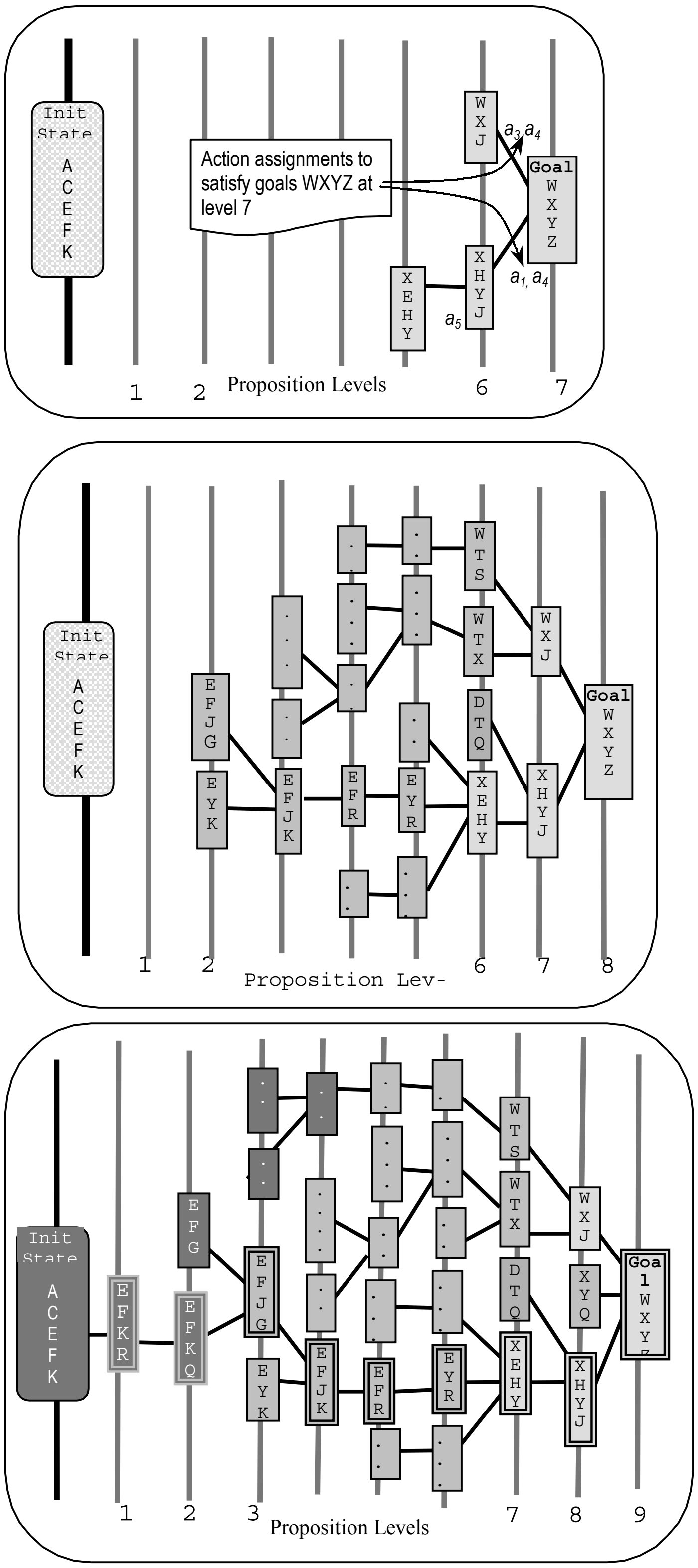


Figure 4. *Graphplan's regression search space: 3 consecutive episodes*

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.

2.2 Graphplan cast as state space search

If we adopt a higher level, more abstract view of Graphplan's search process it can be viewed as backward-directed state space 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. 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 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. For simplicity, the 'no-good' states of Figure 3 are not shown in this state-space trace.

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 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. If we adopt the state space view of Graphplan's search (Figure 4), we find that within a 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*** :the distance in terms of associated planning graph levels between the state generated in Graphplan's regression search and the initial state[1]

g :the cost of reaching the node-state from the goal state in terms of number of CSP epochs (i.e. the difference between the number of 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, that guarantees 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.

[1] 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 we of primary interest to us.

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

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. As would be expected for 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 "replay" 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*.[2]

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 our running problem example. This implies 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 (marked with a darker shade). 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 investigated the application of additional memory to store a trace of the explored search tree. The first implemented approach, EGBG (described in the following subsection), sought to leverage an appropriately designed search trace to avoid as much of the inter-episode redundant search effort as possible (Zimmerman and Kambhampati, 1999). This search trace exploits the following 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

[2] 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 and 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.

active at level k+1, but once it becomes inactive at a level it never gets re-activated at future levels. For example, when a new action *a1* is introduced at level k it's mutex status with every other action (in pair-wise fashion) at that level is determined. If it is mutex with *a4* the pair may eventually become non-mutex at a future level, but thereafter they will remain non-mutex. If *a1* is initially non-mutex with *a3* at level k it will never become mutex at higher levels.

- Two actions in a level that are "statically" mutex (i.e. their effects or preconditions conflict with each other) will be mutex at *all* succeeding levels.
- 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.

3.1 Aggressive use of memory in tracing search: the EGBG planner

The characteristics listed above 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. To this end, we offer two observations:

Observation 3.1) *The intra-level CSP-style search process conducted by Graphplan on a set of propositions, S_n, at planning graph level k+1 in episode n+1 is identical to the search process on S_n at level k in episode n as long as the following two conditions hold:*

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

Observation 3.2) *The trace of Graphplan's search episode n+1, initiated on a set of propositions S, at planning graph level k+1, is identical to its episode n search on S at level k as long as the following two conditions hold:*

1. *The two conditions of observation 3.1 hold for* ***every subgoal set*** *(state) generated by Graphplan in the episode n regression search on S at level k.*
2. *For every subgoal set S_n at 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 at the time S_n is generated in episode n+1. Conversely, for every subgoal set S_n 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_n 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 *k* in episode *n*. If that search failed to extract a solution from the k-length planning graph (i.e. reach the initial state), then if it is possible to extract a solution from the *k+1* length graph it must be true that one or more of the conditions of observations 3.1 or 3.2 fails to hold for the episode *n* search trace. Based on this, our most aggressive approach to search tracing (implemented in the EGBG planner), uses its search trace from episode *n* to direct search in a sound and complete manner during episode *n+1* by:

1. Ensuring that the relevant mutexes (i.e. the *dynamic* mutexes) and actions of observation 3.1 get checked for each subgoal set of observation 3.2.
2. Ensuring that the states that matched a cached memo in episode *n* are checked against the memo cache at the next higher level in episode *n+1*.

For each condition 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 will simplify a more detailed description of conducting search using the PE:

Search segment: 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 search segment S_n, generated a 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 action consistency checking results during the attempt to satisfy the subgoals in the previous search episode

Thus, a search segment represents a state plus some path information, but we may use the terms interchangeably. All the proposition lists appearing in boxes in Figure 4 are search segments. The procedure used by EGBG to build and use these search segments is outlined in the sidebar below and the pseudo code will be discussed in the next subsection.

Pilot explanation (PE): The search trace, consisting 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 plan graph level *k, k+1, k+2*, etc. We also 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.

Hereafter we refer to a PE search segment that is visited in the solution episode 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*.

Given these definitions, we note that the PE, after a search episode *n* on plan graph level *k* is a loose *lower bound* [3] on the set of states that will be visited when backward search is conducted in

[3] It is possible for Graphplan's memorizing process to preclude some states from being regenerated in a subsequent episode. See footnote 1 for an brief explanation of conditions under which this may occur.

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 Tracing the intra-segment CSP search

As noted elsewhere, the process Graphplan uses to assign a consistent set of actions to a subgoal set is essentially CSP search. Observation 3.1 describes the minimum aspects of the search that must be recorded during search on a search segment Sn (subgoal set) at planning graph level *k* in episode *n*, in order to avoid provably redundant effort in the subsequent episode at level k+1. We would like to conduct search only under variables with newly extended value ranges (i.e. search segment propositions that have at least one new establishing action at level k+1) and at points in the level k search that backtracked due to attempts to assign values that violate a 'dynamic' constraint (i.e. two actions that are dynamic mutex at level k). All other assignment and mutex checking operations involved in satisfying the goals of S_n are static across search episodes.

We experimented with several search trace designs for capturing key decision points. The design adopted for EGBG employs a sequence of bit vectors representing assignment results at each test for action set consistency during the k-level subgoal search. The trace uses two bits to represent four decisions/conditions that permit efficient action assignment replay: dynamic mutex, static mutex, no conflict, and a complete, consistent set of assignments that is rejected at the next level due to a memoized no-good. As long as the *order* of actions appearing under the "establishers" list for a proposition remain constant, the bit vectors can be used to replay the search in the next episode on the next higher planning graph level.

The two key conditions described in observation 3.1 are tested as follows: 1) The vectors dictate mutex status checking on *only* action assignments that previously backtracked due to dynamic mutexes 2) New establishing actions for a subgoal are tried after all other establishers are replayed. All other mutex checking associated with search on the search segment's subgoals in previous episodes is avoided; static mutex action assignments and the consistent (non-mutex) assignments are replayed *without rechecking* action mutex status. When a dynamic mutex no longer holds or a new establishing action comes up for assignment, the bit vector is modified accordingly and EGBG resumes Graphplan's CSP-style search, adding assignment vectors to the search segment in the process.

3.3 Conducting search with the EGBG search trace

The high level approach adopted by EGBG in building the initial pilot explanation during the first regression search episode and then using it in subsequent search, is presented in Figure 5 in pseudo code form. In the first episode of regression search on the planning graph an augmented version of Graphplan's 'assign-goals' routine is used to build the search trace (PE) as it attempts to reach the initial state. This routine traces its search progress as described in the previous subsection and as outlined in the ASSIGN-GOALS pseudo code of Figure 6.

If no solution is possible on the k-length planning graph, the graph is extended and EGBG then uses its ASSIGN-SEG-GOALS routine to replay key features of its previous search as captured in the PE, for all subsequent search episodes. In the process, the PE is augmented according to the search space visited. The pseudo code outlining this routine appears on the right side of Figure 6. Figures 5 and 6 depict two processes, key to employing the search trace, that merit further discussion;

PE transposition: In the 'for' loop of EGBG-PLAN (Figure 5) a particular planning graph level, *n*, is associated with each PE level in each search episode. This corresponds to *transposing* up one planning graph level, the pilot explanation of search segments (states) generated/updated in the previous episode. That is, for each search segment in the PE associated with a planning graph level *j* after search episode *n*, associate it with level *j+1* for episode *n+1*:
1) $\forall S_i(n,j) \in PE(n) : S_i(n,j) \xrightarrow{assoc} S_i(n+1, j+1)$

Visiting a search segment: This is implemented by the ASSIGN-SEG-GOALS routine of Figure 6. For segment S_i at plan graph level *j+1*, visitation is a 4–step process:

1. Perform a memo check to ensure the subgoals of S_i are valid (not a no-good) at level *j+1*

Figure 5. *EGBG pseudo code for top level search*

```
EGBG-PLAN (problem)
    Build planning graph PG until first level, k, for which problem goals, g, are non-mutex
   ..conduct Graphplan-style search on g & build initial PE -
    Create new search segment SS holding goals g
    If ASSIGN-GOALS (SS, g, nil, k) returns a search segment,
    SS_0  --SUCCESS;
     Extract solution from linked PE search segments, beginning
     with SS_0 at level 0.  -- DONE.
   Else no k-length solution possible;
 ▪  L1: Extend planning graph: k<-k+1
 ▪   -Use PE to direct subsequent search-
      For  p = number of deepest level of  PE to top level (0)
           n = planning graph level associated with PE level p
              = k - p
         L2:  Select an unvisited search segment, SS in level p
              Let SSgoals = subgoals of SS
                   SSresults all ordered, goal-by-goal result vectors from SS  (action assigns for SS
                   subgoals from previous episode)
              Clear the assignment vectors from SS
              If SSgoals does not match a memo cached at
                 level n of PG, then:
                 If ASSIGN-SEG-GOALS (SS, nil, SSgoals,
                    SSresults,  n ) returns a search segment, SS_0
                    --SUCCESS;   Extract solution from the
                     linked PE search segments, beginning with
                    SS_0 at level 0.
                    DONE
                 Else Return to L2
 ▪  When all search segments in all PE levels visited:
    Done with PE processing on k-level planning graph
    Go to L1
```

2. 'Replay' the previous episode's action assignment sequence for all subgoals in S_i, using the segment's ordered assignment vectors. Conduct mutex checking on *only* those pairs of actions that were *dynamic* mutex at level j. 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. (A child search segment is created, linked to S_i, and added to the PE whenever S_i's goals are successfully assigned, entailing a new set of subgoals to be satisfied at level *j*.)

3. For each S_i subgoal in the replay sequence, check also for new actions appearing at level *j+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 j+1 if no solution is extracted via the assignment/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.

Figure 6. *Pseudo code for EGBG's regression search routines*
Left -*new search on the planning graph* Right -*search replay using the search trace*

Regression search on a new subgoal set while building the search trace (PE)

```
ASSIGN-GOALS (SS (search segment) SSgoals
  (subgoals left to assign), A (actions already assigned),
  k (PG level) )
  If SSgoals is empty or k is 0 (the initial state) then:
        SUCCESS: Return SS to EGBG-PLAN.
  else there are goals left to satisfy:
      Pop front goal, g, from SSgoals
      Append empty assignment results vector,
       gresults to SS vectors list
      Let Ag = the set of actions from level k of PG
      that support g
      L1: For each action act from Ag
        If act is dynamic mutex with an act in A:
           Append 'dyn mutex' to gresults vector
        else if act is static mutex with an act in A:
           Append 'stat mutex' to gresults vector
        else act has no conflict with actions in A:
          Add act to A (assigned acts)
          If g is the last goal for SS, then:
           -prepare for search at lower level on
            $SS_1$goals, the subgoals of SS regressed
            over A's assigned actions -
            If $SS_1$goals matches a memo at PG level
              k-1 then: (we backtrack on nogood..)
               Append 'no-good' tag to gresults
            else ( move down to next PG level)
             • Create child search segment $SS_1$
               linked to parent SS, holding sub-
               goals $SS_1$goals, and A (assigned ac-
               tions)
             • Append 'no-conflict' to gresults
             • Add $SS_1$ to PE level associated with
               PG level k-1
             • Call ASSIGN-GOALS ($SS_1$,
               $SS_1$goals, nil, k-1)
             • Memoize $SS_1$goals at PG level k-1
          else g is not the last subgoal for SS:
           • Append 'no-conflict' to gresults
             - move down to assign next subgoal -
           • Call ASSIGN-GOALS (SS, SSgoals,
                                     A, k-1)
      Loop to L1.
    Return from ASSIGN-GOALS
```

Replaying regression search captured in search trace (PE)

```
ASSIGN-SEG-GOALS (SS (search segment), A (actions
   already assigned), SSgoals (goals left to assign), SSresults
  (remaining assign results), k (PG level) )
 If SSgoals is empty, A is a consistent set of actions satisfying
   the goals of SS: Return.
 If SSgoals is not empty then:
      Pop front goal, g, from SSgoals
      Pop front assign result vector, gresults, from SSresults
   Let Ag = the set of actions from level k of PG that support g
    -replay assignments for g from previous episode-
    L1: For each assign result, ares, in gresults vector
      • Pop action act from Ag
      • If ares indicates no conflict with act, then
         • Add act to A (assigned acts)
           - move down to next goal-
         • Call ASSIGN-SEG-GOALS (SS,A ,SSgoals,
                                      SSresults, k)
        else if ares indicates act was static mutex;
           no need to retest, loop to L1
        else if ares indicates act was dynamic mutex
          with an act in A, check current mutex status:
          If act is still mutex, loop to L1
           else it's no longer mutex with any action in A:
           • Change ares to 'no conflict' in gresults
           • Add act to A (assigned acts)
             - resume backward search and extend PE-
           • Call ASSIGN-GOALS (SS, SSgoals, A, k)
        else if ares designates a no-good from previous
          episode then memo-check regressed subgoals
          for A at level k-1:
          If subgoals for A are also no-good at level k-1,
            loop to L1
           else regressed subgoals are no longer no-good;
           • Change ares to 'no conflict' in gresults
           • Create new search segment $SS_1$ with goals
             based on regressed goals of A's assigned
             actions
           - resume backward search at level k-1, extend PE-
           •Call ASSIGN-GOALS (SS, $SS_1$goals, nil,
                                            k-1)
    When all assignments in gresults have been replayed;
     augment vector with results of any new establishers for
     subgoal g that first appeared at level k:
    L2: If Ag is empty, Return from ASSIGN-SEG-GOALS
        else attempt to assign new action
        • Pop action act from Ag
        • If act is mutex with an action in A then;
           o Append assign result (stat or dyn mutex) to
             gresults
           o Return to L2.
          else assign action & resume backward search
           o Add act to A (assigned acts)
           o Call ASSIGN-GOALS (SS, SSgoals, A , k )
           o Return to L2.
```

The search algorithm for EGBG essentially alternates the selection and visitation of a promising state from the search trace of its previous experience, with a focused CSP-type search on the state's subgoals. The latter process seeks to focus on only those aspects of the previous search that could possibly have changed and areas of the search space not previously explored. The former process suggests that 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. We might, for example, 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). Intelligent traversal of the state-space view of Graphplan's search space is taken up in Section 5, where we argue that this is, perhaps, *the* key advantage afforded by such the search trace. The desire to exploit this freedom more fully helps motivate the move to PEGG's more sparse style of search trace from the detailed trace used by EGBG.

As it turns out, when the search segments in the PE are visited in any fashion other than either top-down or bottom-up order (in terms of PE levels), the extensive intra-segment tracing conducted by EGBG greatly complicates the bookkeeping and can incur significant memory management overhead. 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 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 better performance was generally observed when *h* was weighted by a factor of 5 relative to the *g* component of the heuristic *f-value*. However, unlike these state-space planning systems, which adopt this as their primary heuristic, the guarantee of plan optimality for EGBG does not depend on the admissibility of this secondary heuristic. 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.

The next section reports the performance of a version of EGBG that implements the approach described above, with little attention to the issue of memory management. The memory efficiency aspect of using a search trace is addressed in section 4.

3.4 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[4] (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. 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 memory demands for systems such as EGBG.

We considered a variety of methods with respect to this issue, and discuss here a suite of seven that *together* have proven instrumental in helping EGBG (and later, PEGG) overcome memory-bound

[4] The values have been updated to reflect performance on the same machine used for other experiments in this study and reflect some changes in the tracking of statistics. All table results in this study are for code compiled in Allegro Lisp version 6.0 for MS Windows, on a 900 Mhz laptop with 384 MB RAM memory.

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 transition to a bi-partite planning graph. Four of the six most effective methods are *speedup* techniques from the CSP field, however our interest lies primarily in their impact on search trace memory demands. The seventh method is a novel variant of variable ordering, which we call 'EBL-based reordering', that takes advantage of the fact that we are using EBL *and* have a search trace available. While this method is readily implemented in PEGG, the strict ordering of the assignment vectors employed 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.

The manner in which the first six techniques are employed in the context of me-EGBG (and PEGG) is outlined below. There are two major modes in which they can impact memory demand; 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). We will compare the impact of the six methods along these dimensions, after summarizing the application of each method to building and searching on a planning graph using the search trace.

Domain preprocessing and invariant analysis:

The speedups attainable through preprocessing of domain and problem specifications are well documented (Fox and Long, 1998a, 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. We have focused the domain preprocessor for me-EGBG /PEGG on identification and extraction of invariants in action descriptions, including typing, and subsequent rewrite of the domain in a form that is efficiently handled by the planning graph build routines. We also discriminate 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 the 'replay' of action assignments when a search segment is visited.

As we will show, 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 address 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.

<u>*Explanation Based Learning and Dependency Directed Backtracking*</u>*:*

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 defer 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 and goal assignments that have resulted in backtracking. This frees the search from chronological backtracking, allowing search to jump back to the most recent variable taking part in the conflict set. The conflict set that is eventually regressed back to the first goal after completing all attempts to satisfy a subgoal set also represents a valuable 'minimal' no-good for memoization. This memo is usually shorter and hence more general than the one generated and stored by standard Graphplan. In addition, an EBL-augmented Graphplan generally has smaller memo caches in terms of memory.

Both EGBG and PEGG have been outfitted 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 assignment vectors (and parts of vectors) whenever the conflict set for a new episode dictates a replay order that differs from the previous episode.

Less obvious than their speedup benefit perhaps, is the role EBL and DDB can 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. Since a search trace grows in direct proportion to the search space actually visited, such techniques that prune search can greatly reduce memory demands for systems such as EGBG or PEGG.

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.[5] Both of these are described in Section 5 where the heuristics used to direct the traversal of the PE states is discussed. As discussed below, we have found the benefits of distance-based variable and value ordering for our search trace-based planners to be highly problem-dependent, both in terms of memory reduction and speedup (in some cases they can even slow solution search). Their effectiveness also varies considerably with the particular ordering heuristic used on a problem.

The use of memory by EGBG/PEGG to build and maintain the planning graph and search trace structures provides added benefits in reducing 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

[5] 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.

heuristic, so that the ordering only has to be computed once.[6] 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 stands in contrast to 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.

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)*.

4.1 Impact of enhancements on EGBG memory demands

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. There is no single configuration of techniques that proves to be optimal across a wide range of problems. Since there is a computational overhead associated with these methods, it is generally possible to find a class of problems for which performance *degrades* due to the presence of the method when compared to a planner configuration that doesn't use the technique. We have settled on this set of techniques based on their joint impact on the me-EGBG / PEGG memory footprint over an extensive variety of problems.

Before reporting the runtime impact of the techniques on EGBG, we characterize them according to the two memory impact modes mentioned at the beginning of this section, 1) reduction in PE size and 2) reduction in space required for planning graph or 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. The Figure 7 plot illustrates for each method, the relative degree of memory reduction impact relative to these two facets, *when the method operates in isolation of the others*. (We defer discussion of the EBL/ Reorder method to Section 5.) As we've noted previously, the actual impact on both memory and speed of these techniques is highly dependent on both the problem and which of the other methods are active.

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

[6] 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. However, it is actually made more difficult when a bi-partite graph is used, since the action ordering under a given proposition will be used across all levels of the planning graph. For action assignment efficiency reasons, it is preferable to have these actions ordered according to level in which they first appear. This is essentially the 'set level' ordering, and it is by far the lowest cost action ordering heuristic for a system using a bi-partite planning graph

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 have great consequence in tower-of-hanoi and some logistics problems.

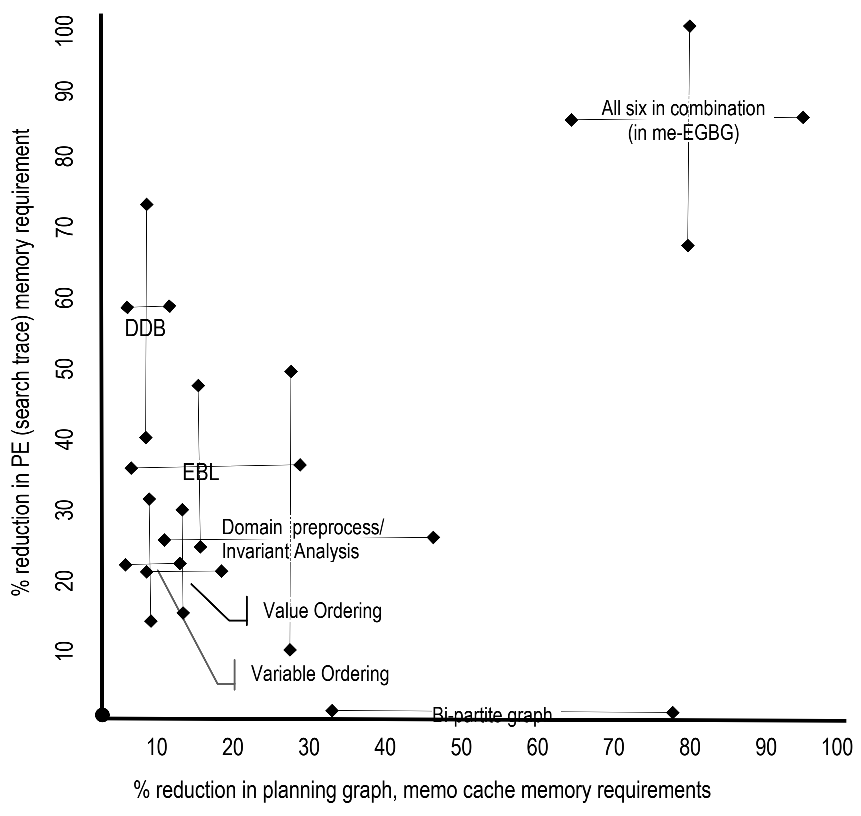


***Figure** 7: Memory demand impact along two dimensions for six memory reduction/speedup techniques when applied independently and as a suite (within EGBG).*

Evidence that the six methods outlined in the previous section, combined, can compliment each other is provided 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 category exceeds 80%. No single method averages more than a 55% reduction along these dimensions in isolation.

The next section compares the performance of EGBG and EGBG augmented with the above enhancements to a similarly augmented version of Graphplan.

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

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 (Graphplan enhanced with the techniques of the previous section), and the two versions of EGBG are compared across 36 benchmark problems in a wide range of domains, including problems from all three AIPS planning competitions held to date. 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 the first version. Of the 36 problems, standard Graphplan solves 11, the original EGBG solves 12, GP-e solves 28, and EGBG with these augmentations solves 27. Where me-EGBG and GP-e solve the same problem, me-EGBG is faster by up to a factor of 62x, and averages >5x speedup. Relative to *standard* Graphplan, on the eleven problems it can solve, me-EGBG exhibits speedups 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), the 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 its strength lies in using the PE to shortcut Graphplan's episodic search process, its efficiency advantage appears only for problems with multiple search episodes and a high fraction of runtime devoted to search. Thus, no speedup (or a small slowdown) relative to GP-e 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 where the problem goals are present and non-mutex.

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 heuristic state-space planners observed in biasing towards low *h-value* states (Bonet and Geffner, 1999, Nguyen and Kambhampati, 2000), translates to some extent to search on the planning graph.

Problem	Graphplan		EGBG		me-EGBG (memory efficient EGBG)		SPEEDUP (me-EGBG vs. GP-e)
(steps/actions)	cpu sec **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.*

'Standard Graphplan': Lisp version by Smith and Peot. GP-e and memory efficient EGBG employ a bi-partite planning graph, domain preprocessing, EBL/DDB, "adjusted sum" goal ordering, "level-based" action ordering.
"Size of PE" is final search trace size in terms of the number of "search segments"
'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 weaknesses in its approach to search trace directed planning:

1. The intra-segment action assignment 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. In spite of its surprising effectiveness in many problems, 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 problem goal state. In the search trace parlance, the only seed segment in the PE is the topmost search segment, so bottom-up visitation of the PE states is more costly than Graphplan's top-down approach. A more flexible and informed traversal order is indicated here.

The first shortcoming is manifest in classes of 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). Due to the overhead associated with building its search trace, EGBG takes a particularly hard hit compared to, for example, Graphplan. Another result of using the assignment vectors is that although EGBG employs EBL/DDB when it conducts new Graphplan-style search, it does not exploit the speedup technique during the action assignment replay of a search segment, due to the difficulty of dynamically updating the assignment vectors.

An obvious tact to address the second shortcoming is to traverse the search space implicit in the PE according to state space heuristics. Unfortunately, EGBG incurs significant processing overhead associated with visiting the search segments in any order *other than* bottom up. 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.

We next consider a different architecture as well as strategy for employing a search trace. The PEGG planners that lie along the right branch of the Figure 1 family share this approach, which trades off the chronicle of action assignment vectors for a more skeletal and flexible structure, and shifts its focus to informed traversal of the state space implicit in the search trace.

5 Tracing at the state level and focusing on the state space view: The 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. The important 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. 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 principle 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 take a global view of the search space visited in previous episodes, and are able to employ a variety of state-space heuristics to preferentially visit promising regions of the search space.

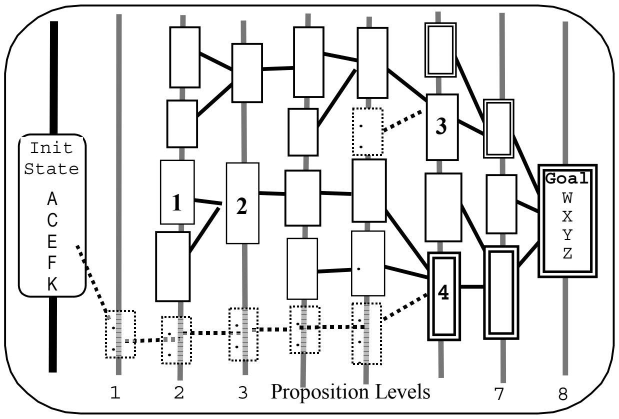


Figure 8. *The PE for the final search episode of a hypothetical problem. Search segments in the PE at the onset of search appear in solid lines, double lined boxes are plan segments, dashed lined boxes indicate states newly generated in regression search during the episode. Visitation order according to secondary heuristic is indicated.*

Following the first search episode, the PEGG planners have available a search trace providing a concise state space view of their search space. A planner with freedom to traverse this search space in any desired order has a clear advantage over one with fixed-order traversal, at least in the episode in which a solution can be found. Figure 8 depicts a small hypothetical search trace in the final search episode in order to illustrate this advantage. Here search segments in the PE at the onset of the episode appear in solid lines and all plan segments (i.e. states that can be extended to find a valid plan) are shown as double-lined boxes. Note that typically there may be many such latent plan segments in diverse branches of the search trace at the solution-bearing episode, and the figure reflects this. 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). In this section, we describe our investigation of heuristics to direct this traversal of the PE search space for the PEGG planners. Looking ahead, we note that as long as we are constrained by Graphplan's policy of exhaustively searching the planning graph level-by-level (as so-PEGG does), we must eventually visit every search segment in the PE during each search episode, and any advantage of heuristic-guided traversal is realized only in the final episode. As discussed in Section 5.4, PEGG (with beam search) relaxes this constraint and the advantage is thereby extended to all intermediate search episodes.

The pseudo-code of PEGG's algorithm is given in Figure 9, and it's apparent that visitation of such search segments is notably simpler than that depicted for EGBG (Figure 5). It now consists of performing a memo check on the subgoals of S_n and conducting Graphplan's CSP-style search on an ordered list of the segment's subgoals. A child segment is created and linked to S_n (extending the PE) whenever S_n's goals are successfully assigned.

The PEGG-PLAN routine combines both state-space and CSP-based aspects in its solution search:

- Choose 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.

- Expand the selected state 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, in that traversal of the state space in the PE is no longer constrained to be bottom-up, level-by-level.

Search trace memory management issues resurface again once we stray from bottom-up traversal, but we defer addressing this in favor of first discussing the development and adaptation of heuristics to search trace traversal.

5.1 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 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 focused on employing a heuristic from the latter work as a *secondary heuristic* to direct PEGG's traversal of its search trace states. Note that the primary heuristic is the planning graph length that is iteratively deepened (Section 2.2), so the guarantee of plan optimality for the PEGG planners does not depend on the admissibility of this secondary heuristic.

There are several key differences between heuristic ranking of states generated

Figure 9. *PEGG pseudo code*

```
PEGG-PLAN ( problem)
  Build planning graph PG until first level, k, for which
  problem goals, g, are non-mutex
 - conduct Graphplan-style search on g & build initial PE -
  Create new search segment SS holding goals, g, ordered
  according to variable ordering method
  If ASSIGN-GOALS (SS, g, nil, k) returns a search segment, $SS_0$  --SUCCESS;
      Extract solution from linked PE search segments,
       beginning with $SS_0$ at level 0.
      DONE.
   Else no k-length solution possible;
    -Use PE to direct subsequent search-
    o  Assess state space heuristic value for new search
       segments (optionally reassess existing segments)
    o  Merge-sort new search segments (states) into the
       PE linked-list, according to heuristic value
    o  Transpose index for all search segments in the PE
       up one planning graph level.
    o  L2:  Select the first ranked segment, SS, from PE
       that has not yet been visited (opt threshold: If the
       f-value of this segment fails to meet heuristic
       threshold criteria, skip remaining PE segments)
    o  Let n = planning graph level associated with the PE
       level of SS
       ▪ If SS is indexed to a planning graph level that
         doesn't yet exist, extend the graph $k \longleftarrow k+1$
       ▪ Perform a memo check on the segment subgoals.
         If they are no-good at level n, remove SS from PE:
           Return to L2.
         else  -visit search segment SS -
           If ASSIGN-GOALS (SS, nil, n ) returns a search
             segment, $SS_0$  --SUCCESS;
             ▪ Extract solution from the linked PE search
               segments, beginning with $SS_0$ at level 0.
             ▪ DONE
           else a subset of SS goals was returned  (the
               conflict set):
             ▪ Memoize returned goals
             ▪ Return to L2

ASSIGN-GOALS ($SS_J$ (search segment)  $SSgoals_J$ (subgoals left
to assign), A (actions already assigned),  n (PG level) )
 If n is 0 (the initial state) then:
    o SUCCESS: Return $SS_J$ to PEGG-PLAN.
 else if $SSgoals_J$ is empty,
  - initiate regression search on level n-1 -
    o  Regress $SS_J$ goals over A actions to get $SSgoals_K$
    o  Create new search segment $SS_K$ holding $SSgoals_K$ goals
    o  Order goals according to variable ordering heuristic
    o  Call ASSIGN-GOALS ($SS_K$, $SSgoals_K$, nil, n-1 )
    o  Memoize returned conflict goals at PG level n-1
    o  Reorder $SSgoals_K$ according to conflicts returned
    o  Return
 else there are goals left to satisfy:
   ...Assign consistent set of actions for remaining goals in
   Graphplan's fashion, augmented by DDB and EBL tech-
   niques...After each goal is assigned a valid action, ASSIGN-
   GOALS is called on remaining goals
```

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, due to the iterative search process, typically must consider whether to *revisit* a state in many consecutive search episodes. Ideally, a heuristic to rank states in the search trace should reflect level-by-level evolutions of the planning graph, since the transposition process associates a search segment with a higher level in each successive episode. For each higher planning graph level that a given set of propositions 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.

Another important difference between ordering states in a state space planner's queue of previously *unvisited* states and ordering of states from the search trace, is that in the latter case the set of states also includes 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 8, we desire a heuristic that can, for example, discriminate between the #4 ranked search segment and the top goal segment (WXYZ). In a sense we would like the heuristic measure for a state to *discount* the value associated with any children already present in the trace, so that only the potential for it to generate new promising search branches is considered.

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.

5.1.1 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). One of the most easily accessible, given a planning graph, is the notion of 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, we found it 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 mentioned in the discussion of memory efficiency improvements (Section 4) we use it as the default heuristic for value ordering *as the graph is constructed*, due to the combination of its low computational cost and its synergy with building and using a bi-partite planning graph. An action a_1 first appears in the graph at *lev(Prec(*a_1*))*, that is, when its preconditions are first present and non-mutex.

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 inexpensive to calculate 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) $$h_{adjsum}(S) := \sum_{p_i \in S} h(p_i) + (lev(S) - \max_{p_i \in S} lev(p_i))$$

It is essentially 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 first 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.). One approach that addresses positive interactions is the so-called 'relaxed plan' distance-based heuristics, and several studies have demonstrated the power of this genre for backward and forward state-space planners (Nguyen and Kambhampati, 2000, Hoffman, 2001). However, as reported in the former, the primary beneficial effect of this metric when incorporated in 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 on the planning graph ensures optimal make-span there is little incentive to incur the expense of the relaxed plan calculation.

5.1.2 Specializing a heuristic for the search trace

The negative interactions expression in the adjusted-sum heuristic suggests a straightforward means of using PEGG's search experience to dynamically improve the estimate. The *lev(S)* term represents the planning graph level at which the subgoals are *binary* non-mutex with each other. However, once regression search on *S* at graph level *k* is complete (and fails) in a given episode, the search process has essentially discovered an n-ary mutex condition between a subset of the goals in *S* at level *k.* At that point we can conservatively update the estimate of *lev(S)* to be *k+1.* This achieves some of the flavor of a desired heuristic for ranking search trace states; Given two states, an ancestor state and one of its descendents, that would otherwise have the same adjusted-sum f-value, the *lev(S)* update will effectively favor the descendent, i.e. it is biased against the state that has been visited the most and failed to extend to a solution. This constitutes a simple approach for boosting our heuristic estimator based on search experience, and the PEGG planners use it by default.

We now turn our attention to the issue of 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 (and indeed any of the distance-based heuristics) to possible differences in the implicit regression search space 'below' a set of propositions, *S*, at some level *k* versus level *k+1.* Assuming the propositions are present and binary non-mutex with each other at level *k* (necessarily the case if *S* is a state generated in regression search), only the cost summation factor in equation 5-3 could conceivably change when *S* is evaluated at level *k+1.* This could occur only if a new action establishing one of the *S* propositions, say a_j, appears for the first time at level *k*+1 and the sum of a_j*'s* precondition subgoal costs is *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[8].

In contrast to the tendency of conventional distance-based heuristics to be static, we desire a heuristic that will 'improve' (i.e. reduce) a state's h-value when it's transposed to a graph level at which new branches of regression search might finally reach the initial state. Conversely, when the planning 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

[8] 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.

above level-off. An effective heuristic for ordering segments in the search trace might exhibit a significant advantage over distance-based heuristics if it ranks states in static levels low in the queue when appropriate.

We investigated a variety of alternatives for hybrids of the distance-based heuristics that incorporate factors reflecting 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 work was guided by Observations 3.1 and 3.2 of Section 3.1, which describe the three planning graph and memo cache properties that determine whether regression search on a subgoal set *S* will change 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).

For convenience 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). As noted previously, 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 it's current level and propagating an appropriately weighted measure up to its parent, we can compile a good estimate of flux for the factors 1 and 2 above.

The third flux factor above is perhaps the most unwieldy and costly to estimate; the 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.* (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 contribute to the conflict set used to direct search within *S* whenever such backtracking occurs.)

Combining the two flux measures that track new actions with the largely static adjusted-sum distance-based heuristic, we define a heuristic that is sensitive to the evolution in search potential as a state is transposed to higher planning graph levels:

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

5-4) $$h_{adjsum-flux}(S) := \sum_{p_i \subset S} h(p_i) + (lev(S) - \max_{p_i \in S} lev(p_i)) \; - w_1 * \frac{\sum_{p_i \subset S} newacts(p_i)}{|S|} - w_2 * \sum_{s_i \subset S_c} childflux(s_i)$$

adjusted-sum terms minus flux terms

where: p_I is a proposition in state S

newacts(p)

$newacts(p_i)$ 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 of child state s_i

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 adjust the h-value bias towards either the distance-based component or the measure of potential new search. Experimentally, over a broad range of problems, PEGG's performance degrades when the flux factor contribution dominates the h-value and improves over the pure adjusted-sum heuristic when they constitute between 0 and 30% of the h-value. Optimal weighting values are, not surprisingly, 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.

Evidence that the adjsum-flux heuristic outperforms adjusted-sum in so-PEGG is provided in Figure 10. The chart compares the heuristics 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. 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. Examination of the

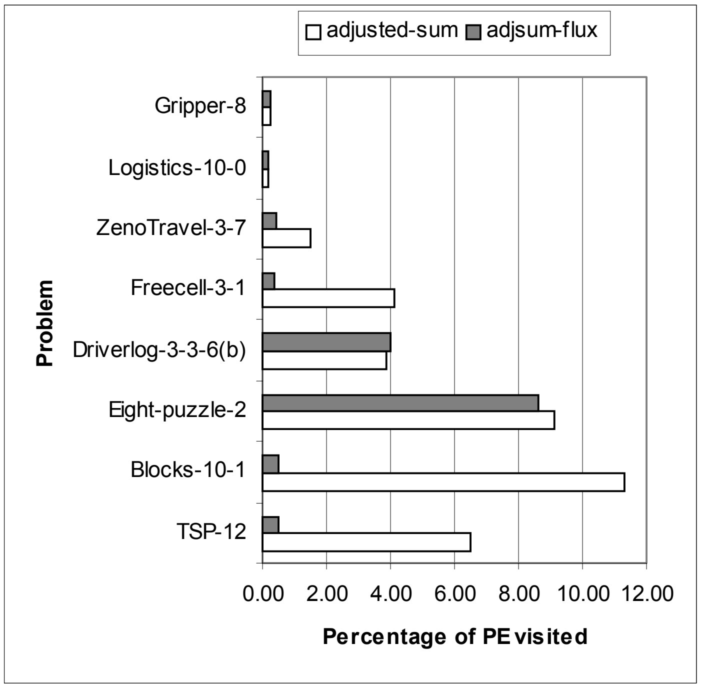


Figure 10. *Comparison of heuristic accuracy in selecting a seed segment from the PE in the solution search episode*

PEGG's search history show 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 a large percentage of the states represented in the PE lie in static levels in the later search episodes. 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.

This is the case for the Blocks-10-1 and TSP-12 problems and explains the apparent adjsum-flux advantage of Figure 10.

5.2 Memory management under arbitrary search trace traversal order

Consider again the PE at the time of the final search episode in Figure 8. If we allow the search segments to be visited in an order other than deepest PE level first, we encounter the problem of re-generating states that are already contained in the PE. The visitation order depicted by numbered segments in the figure might result from a fairly informed heuristic (i.e. choosing a plan segment as the 4th search segment to visit), but when followed many states already resident in the PE will be re-generated. This includes, for example, all the as yet unvisited descendents of the third segment in the order. 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. For efficiency in checking for duplicate states, 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 (if the state meets the beam search criteria).[9]

5.3 Learning to order a state's subgoals

Before presenting PEGG experimental results, we describe here one more method that the search trace enables the planner to exploit. Because they employ both EBL and a search trace, the PEGG planners are able to overlay a yet more sophisticated version of variable ordering on top of the

[9] In the interests of simplicity, the Figure 8 pseudo code does not outline the search trace memory management process described here.

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 goals in the state must be assigned an action from the planning graph, it is best to first attempt to assign values to goals that are likely to be 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, through using EBL during search, we have 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 the PEGG planners tightly integrate the various CSP and efficiency methods, and their independent influence cannot be readily assessed.[10] In order to isolate the impact of EBL-based reordering from that of EBL itself in EGBG, the EBL process is activated, but the conflict sets produced are used only in reordering, not during memoization. The average reduction in search trace memory over the 12-problem sample is seen to be about 18% for EBL-based reordering alone. This compares favorably with the 22% average reduction of the distance-based ordering especially since, unlike the adjusted sum ordering, the EBL-based reordering *only takes effect in the 2nd search episode*.

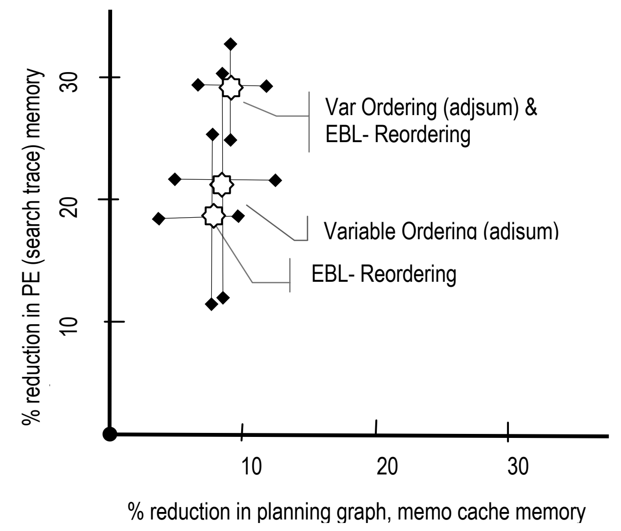


***Figure 11**: Memory demand impact along two dimensions for 'adjusted sum' variable ordering and EBL-based reordering techniques when applied independently and together (within EGBG).*

The plot also reveals that the two modes of ordering are complimentary to a significant extent. Based on tests 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;

[10] Based on the success of the various memory-efficiency methods with EGBG, all versions of PEGG implement them by default. Although an graph analogous to Figure 5 for the PEGG planner would differ in terms of the actual memory reduction values, we are confident that the overall benefits of the techniques would persist, as would the relative benefit relationship between techniques.

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 distance-based heuristic is then used to order goals in the conflict set and this list is 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 reduce 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.

5.3 Experimental results with so-PEGG

We can now characterize the performance of the step-optimal version of PEGG outfitted with the adjsum-flux secondary heuristic and EBL-based reordering. Table 4 compares so-PEGG against GP-e and the memory efficient version of EGBG over most of the same problems as Table 3. (We defer discussion of the 'PEGG' and associated 'Speedup' columns until the next section.) It also includes a variety of larger problems that neither of the latter two planners can handle. Table 3 problems that were easily solved for GP-e and me-EGBG, such as those in the AIPS-98 'movie' and 'mystery' domains, are omitted in Table 4. Focusing for now, only on the GP-e, me-EGBG, and so-PEGG (step optimal) 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, so-PEGG can be expected to 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 systems can solve, so-PEGG slightly under performs me-EGBG. We would generally expect me-EGBG to dominate for problems with multiple search episodes, as long as memory constraints are not an issue, since the skeletal PEGG search trace lacks the information needed to avoid most of the redundant consistency-checking effort. 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 reduced demands in maintaining its more concise search trace. Note that, since both planners visit all states in their PE for search

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 4. *so-PEGG and PEGG vs. Graphplan and enhanced Graphplan*

GP-e: Graphplan enhanced with bi-level PG, domain preprocessing, EBL/DDB, goal & action ordering
me-EGBG: memory efficient EGBG (bi-level PG, domain preprocessing, EBL/DDB, etc.)
so-PEGG: step-optimal, same enhancements as GP-E, search via the PE, traversal according to heuristic
PEGG: bounded PE search, best 50 search segments visited, as ordered by the adjsum-flux state space heuristic

Parentheses adjacent to cpu time give (# of steps / # of actions) in returned solution.
Allegro Lisp platform, runtimes (excl. gc time) on Pentium 900 mhz, 384 M RAM

episodes in which a solution *cannot* be extracted, there is no obvious advantage to prefer one state traversal order over the other in these 'intermediate' episodes.[11]

The next section describes the last evolution of the search trace based planners we report on; an approach that more fully exploits the advantage provided by a search trace in the 'intermediate' search episodes, i.e. those from which no solution can be extracted. This tactic boosts planning speed by an order of magnitude *and* further cuts memory demands tied to the search trace, in some cases even for the planning graph itself.

5.4 Trading off guaranteed step-optimality for speed and reach: The PEGG planner

Many of the large, more difficult problems to solve with Graphplan's IDA* style search have 20 or more such search episodes before the episode in which a solution can be extracted is reached. The cumulative search time associated with these episodes can be a large portion of the total search time, so there is considerable motivation to reduce it. Here we consider an effective alternative to Graphplan's search routine (as well as that of EGBG and so-PEGG), which exhausts the entire search space in each episode up to the solution bearing level. It is, of course, 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. Having a search trace available across all but the first search episode, we investigated possibilities for using it to conduct focused search on a k-length planning graph only until it's deemed unlikely that it contains a solution. We are interested in the extent to which this can be pursued while producing plans with the same or nearly same makespan as Graphplan's solution.

The approach reported here shortcuts the time spent in search during these intermediate episodes by using the distance-based 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 and according to a user-specified limit. The implemented planner, which we call simply PEGG, can be seen as taking another step away from Graphplan's IDA* search process, as it is released from exhaustive search of the planning graph in *all* search episodes. In so doing, PEGG trades off Graphplan's (and so-PEGG's) *guarantee* of finding an optimal make-span solution, and indeed, even completeness. The empirical evidence indicates that the trade-off is worthwhile in virtually all cases, as PEGG exhibits dramatic speedup over GP-e and so-PEGG in *all* search episodes and yet generally finds a step-optimal solution. In addition, we will show that this beam search approach 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.

The primary modification required of the PEGG algorithm in order to implement this beam search is to set a threshold such that only PE states with secondary search f-values lower than this limit are visited. This is indicated in italics at the step labeled *L2* of the Figure 9 pseudo code. The intent is to

[11] 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.

avoid visiting (and optionally, retaining) search segments that hold little promise of being extended into a solution, as predicted by the heuristic used. When the first segment exceeding this threshold is reached on the sorted queue the search episode ends.

Devising a highly effective threshold test is an interesting problem. It must reconcile the competing goals of minimizing search in non-solution bearing length planning graphs, 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 -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 is a wide variety of strategies for conducting beam search on PEGG's search trace. Before discussing experimental results of the particular strategy we found most broadly effective, we touch briefly in the next subsections on two issues that impact the effectiveness of the search process.

5.4.1 Memory management for beam search on the search trace

Search trace memory management issues resurface yet again when we adopt 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. This suggests that our strategy for dealing 'inactive' portion of the PE might further reduce its memory footprint. At first blush, one might be tempted to pursue a strategy with minimal memory footprint by retaining in memory only the active search segments in the PE. However unlike Graphplan, PEGG cannot extract a solution when the initial state is reached by unwinding the complete sequence of action assignment calls in a search episode, since it *begins* its regression search episodes from particular states in any branch of the search trace tree. It 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.

The approach we settled on for this study in fact does not attempt 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 actual states that are 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 (search segments previously generated in regression search) available to be compared heuristically, the more likely that a plan segment will be included in the active PE.

Since PEGG's skeletal search trace reduces the associated memory footprint to where it is a much less critical issue most of our experimental work has adopted the strategy of retaining in memory *all* search segments generated during search and updating their f-values prior to ranking them 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. Empirically we have found that under these conditions search segments often move between active and inactive status in consecutive search episodes.

5.4.2 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 qualified 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, in fact, we find to be the most effective value), the weights become irrelevant. Admittedly the flux metric is indeed just an estimate of the potential for search on a state to generate new search branches, and 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.

5.5 Experimental results with PEGG

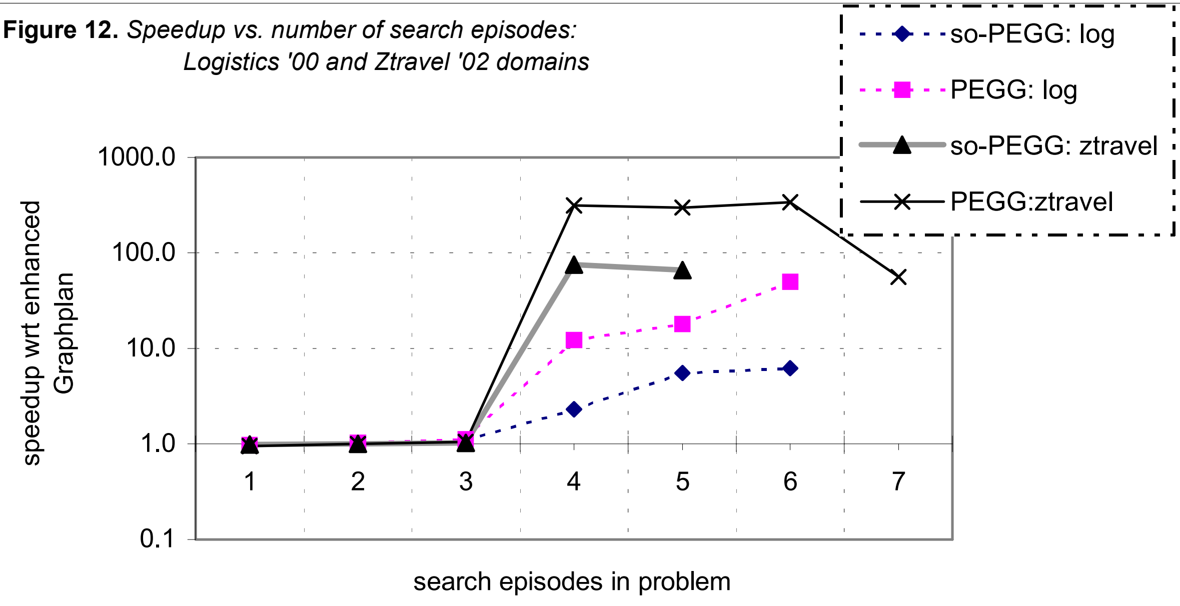
The variety of parameters associated with the beam search approach described above admits considerable flexibility in biasing PEGG towards producing plans of different quality. Shorter makespan plans are favored by more extensive search of the PE states in each episode while heuristically truncated search will tend to generate non-optimal plans more quickly, often containing redundant or unnecessary actions. Here we focus on the former aspect In order to focus on the domain-independent performance of PEGG for these results we fix the various search control parameters on values that were determined to perform well *on average* across a variety of domains and problems. These are as follows:

- o Goal ordering: based on proposition distance as determined by the adjusted-sum heuristic.
- o Value ordering: based on planning graph level at which an action first appears.
- 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.

Returning now to Table 4, the column labeled 'PEGG' reports results generated under these parameters. Clearly, the beam search greatly extends the size of problem that PEGG can be handle; note the fourteen large problems of Table 4 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 problems 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 problems that GP-e fails to solve were conservatively calculated using a GP-e runtime of 1800 seconds.

The problems for which PEGG exhibits the least advantage over GP-e roughly fall into two classes; 1) Those for which the planning graph construction cost constitutes a large fraction of overall runtime 2) Those needing only a couple of search episodes prior to reaching a solution-bearing level. In the first category lie problems like those in the AIPS-98 planning competition 'GRID' domain. The effort spent constructing the planning graph heavily dominates runtime for such problems, so that a system that focuses on major reduction of search time will show little impact. A closer examination of the PEGG runtime trace for each of the problems reveals that the beam search indeed dramatically reduces search time in *every* episode, so much so that for a majority of the problems the cumulative search time has been trimmed to within a factor of 1-2 of the graph construction time.

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



Further dramatic improvement on such problems must rely as much on reducing graph construction costs as search costs.

The second category above draws attention to the particular class of planning problem that PEGG excels at; those in which there are numerous, prolonged search episodes conducted prior to extending planning graph to the solution-bearing length. Figure 12 vividly illustrates this characteristic of the planner by plotting the speedup factors of both so-PEGG and PEGG (under beam search) 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 does not exhaustively search the planning graph in each episode.

5.5.1 Plan quality profile

Turning now to the plan quality side of the coin, we compare the makespan for GP-e and PEGG problem solutions reported in Table 4. As indicated by the annotated steps and actions numbers given in parenthesis next to successful GP-e and PEGG runs[12], the step-optimal produced by enhanced Graphplan is matched by PEGG for all but four of the 36 problems reported. (Problems for which PEGG returns a solution with longer makespan than the step-optimal are ndicated by 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 robust property of PEGG's beam search on the search trace across all problems tested to date.

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, of course. In 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, and Table 5 reproduces those results

Problem	**'N-best'** [state-space search, N=100]	**PEGG** (adjsum-flux heur, 100 best ssegs)	**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 5. *Plan quality comparison of PEGG with N-best beam search for a forward state space planner (Bonet, et. al., 1997)*

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

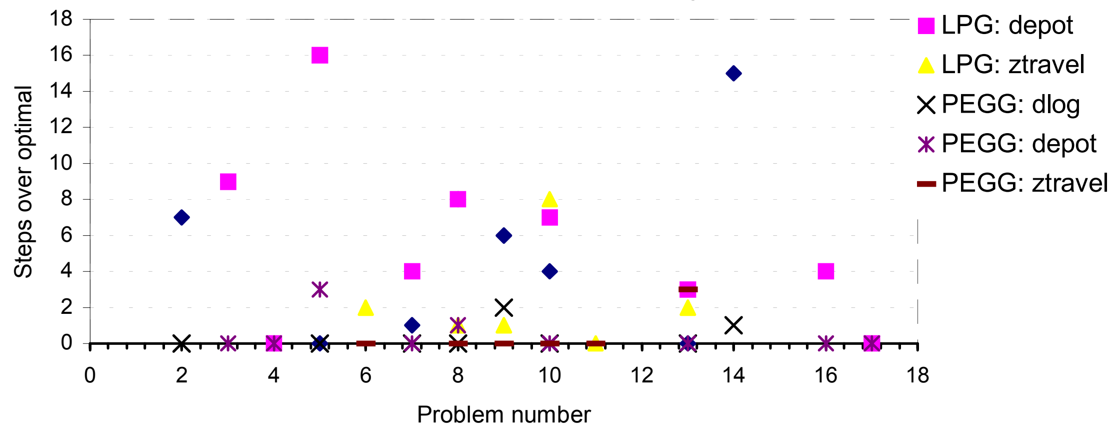
alongside PEGG's performance on the same problems using beam search with 100 search segments visited in each intermediate search episode. The 1997 study compared the N-best first approach against SATPLAN, which produces the optimal length plan, to make the point that the approach could produce plans 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.[13] 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, a 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 13 scatter plot, solution quality for LPG (Gerevini, Serina, 2002) and PEGG are compared against the optimal for 22 problems from just three domains of the 2002 AIPS planning competition. LPG's results are particularly apt in this case since that planner also non-exhaustively searches the planning graph at each level before extending it, although it's 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 clearly excels in this regard, as it's maximum deviation from optimum is three steps, and most of the plot points for its solutions lie right on the optimal makespan axis.

PEGG often finds plans with fewer actions than GP-e for parallel domains and it is interesting to note that this 'Graphplan hybrid' system is not just effective in parallel domains, Graphplan's forte, but impressive in serial domains such as blocksworld as well.

Figure 13: Makespan comparison of PEGG and LPG -Departure from step-optimal plan length



[13] We note only that PEGG's solution times on our platform are over a factor of 10 faster than the SATPLAN results reported in the 1997 study, while producing the same optimal length plan.

5.5.2 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 there 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 6 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 4. AltAlt (Srivastava, Zimmerman, 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 6 reports AltAlt performance for two of the most effective heuristics devel-

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 6. *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 oped for the planner (Nguyen and Kambhampati, 2000), the first of which is the version of the adjusted-sum heuristic that was described in section 5.1.1. Surprisingly, in the majority of problems PEGG returns a parallel, generally step-optimal plan faster than AltAlt returns its serial plan. As a backward state space planner 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 and Related Work

In terms of related work, we will not further discuss here 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. 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. We have endeavored to cite these sources above, where each topic is first discussed. Here we focus on related or alternative strategies for employing search heuristics in planning, generating parallel plans, or making use of memory to expedite search.

Exploitation of a search trace can be seen as directly addressing IDA*'s inadequate use of available memory. The only information carried over from one iteration to the next in standard IDA* search is the upper bound on the f-value. Graphplan's IDA*-style search shares this inefficient use of memory, though it partially compensates with its memo caches that store learned no-goods for use in successive episodes. The search trace goes much further, reducing IDA*'s redundant regeneration of nodes by serving as a memory of the *states* in the visited search space of the previous episode. 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 the EGBG and PEGG planners, they retain as much of their search experience as memory permits to avoid repeating and regenerating node 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's 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 in PEGG (though it would be straightforward to do so) primarily because empirical evidence to date shows that the worst f-value states that could be deleted from PEGG's search trace rarely ever become candidates for expansion before a solution is found via another branch.

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 original idea of employing available memory to record a trace of search experience so as to avoid redundancy in Graphplan's iterative search process has led in unexpected directions. The work evolved such that the resulting planning system, PEGG, no longer derives its primary advantage from learning to avoid unnecessary search effort, but rather from it's ability to exploit strengths of both CSP and state space approaches to search on the planning graph. The search trace essentially provides a state space view of Graphplan's search experience allowing us to investigate the adaptation and augmentation of planning graph based distance heuristics to not only direct the planner's traversal of the search space from a previous episode, but to actively prune that space so as to keep its search focused.

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 arising out of Graphplan's regression search on problem goals and, as such, PEGG exploits reachability information implicit in its planning graph in choosing its n-best nodes for expansion in each episode. 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.

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 common 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, motivated by the large fraction of Graphplan's computation effort spent in consistency checking, 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;

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

where; U_{mt} is the utility of tracking search 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

Essentially, equation 6-1 indicates that amortization of the high overhead of logging consistency-checking experience associated with 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 ineffectual for problems where search in the final search episode generates a large number of states relative to previous episodes and when the only seed segment(s) are at the top or in the first 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 section 3.2). This has its drawbacks, as evidenced by the sixteen problems of Table 4 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 we found 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. It is possible that a less memory-intensive, more flexible, and more efficient algorithm might boost the utility of capturing the consistency checking information in a search trace, but it's not apparent how else to soundly encapsulate the process and still correctly accommodate possible new actions and relaxed dynamic mutexes in subsequent episodes.

The PEGG results of Tables 2 and 3 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.

We note at this point 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 5. 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 meeting the f-value criterion is chosen, Graphplan-style regression search on its 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. The problem with incorporating this into PEGG is that its regression search is greatly expedited by explanation based learning as well as DDB. 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 reasonably informed basis for returning anything other than the full set of subgoals in the state, and returning the entire state generally undermines the entire EBL/DDB process for continued search on the ancestors of the child state.

Experimentally we found that, when PEGG was adapted to enforce the PE state f-value limit on its regression search (returning the entire state as a conflict set), 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 many cases, we traced this directly to the degradation of the EBL/DDB processes, but for other cases, the fault appeared to lie with the difficulty of ranking newly generated states with those already in the PE according to their f-values. The latter problem appears rooted in the complex interaction of two factors that impact the assessed f-value of states in the PE, but not newly generated states;

- The heuristic values of PE states tend to *increase* based on PEGG's use of search experience to improve h-value estimates each time a state is visited (Section 5.1.2)
- The heuristic values of PE states are adjusted *lower* due to the flux adjunct in the adjsum-flux heuristic.

The quality (make-span) of the returned solutions suffered across a broad range of problems. 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 4 results) is rooted in its interleaving of best-state selection from the PE with Graphplan-style depth-first search on the state's subgoals.

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 tradeoff 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 in it can be transposed up any desired number of levels -subject to the ability to extend the graph correspondingly- and have its heuristics re-evaluated at each level. For example, referring back to Figure 4, after the first search episode pictured (top), the XEHY state in the PE could be transposed up from planning graph level 5 to levels 6, 7, or higher, heuristically evaluated at each level, and then be simultaneously compared as distinct states to be visited. Ideally, we'd like to move directly to evaluating XEHY at planning graph level 7, since at that point it becomes a plan segment for this problem (its state is part of a valid plan). If our secondary heuristic can discriminate between the solution potential for XEHY at the levels it can be transposed to, we should have an effective means for further shortcutting Graphplan's level-by-level search process. It seems likely that the flux adjunct will be one key to boosting the sensitivity of a distance-based heuristic in this regard.

This would be a prohibitive idea in terms of memory requirements if we had to store multiple versions of the PE, but 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. Challenges that must be dealt with 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). Interestingly, a real-time version of PEGG more closely related to LRTA* might be based on the method just 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 a version of PEGG that operates in an anytime fashion, seeking to optimize over multiple plan quality criteria. The current version of multi-PEGG (Zimmerman and Kambhampati, 2002) 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.

7 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 a state space view of search experience in the first episode, and using it to guide search in subsequent episodes, dynamically updating it along the way.

The approach melds two of the most effective views of planning, CSP and state space search, and demonstrates that it can exploit strengths of both. Since the planners retain Graphplan's efficient routines for finding consistent sets of parallel actions, we have focused on their ability to speed up production of parallel plans as close to optimal make-span as possible.

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 six 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 transition to a bi-partite planning graph. Beyond their well-known speedup benefits, we have provided experimental evidence of the impressive impact that these methods have on search trace 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 describe the adaptation of distance-based heuristics to support informed traversal of the search space implicit in the PEGG search trace. We then develop an adjunct heuristic we call flux, which augments the distance heuristic with sensitivity to changes in the potential for a search trace state to seed new search branches as it is transposed to higher planning graph levels. Empirical evidence is provided supporting the advantages of employing the flux metric both in the 'adjsum-flux' heuristic and as a filter. We additionally develop some new techniques that leverage the search experience captured in the search trace and demonstrate their effectiveness.

The so-PEGG planner, which like me-EGBG produces guaranteed optimal parallel plans, also 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. The most dramatic evidence of the potential for a search trace guided planner is provided by PEGG. This version 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., Zimmerman, T., Nguyen, X., Kambhampati, S., Do, M., Nambiar, U. Nie, Z., Nigenda, R. (2001). AltAlt: Comjbining Graphplan and Heuristic State Search. In *AI Magazine*, American Association for Artificial Intelligence, Fall 2001.

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. *To appear:* Proceedings of IJCAI-03*,* 2003.