Effective Interplay of State Space and CSP Views in a Single Planner

Terry Zimmerman & Subbarao Kambhampati
Department of Computer Science & Engineering
Arizona State University, Tempe AZ 85287
Email: {zim,rao}@asu.edu

Abstract

AI planning has made impressive advances under several different paradigms of the problem structure and search process. Each view brings its particular strengths to bear on the problem and suffers inherent, relative weaknesses. This study introduces a versatile approach that effectively melds two methodologies used in some state-of-the-art planners; CSP and state space search. The resulting system (PEGG) can operate in a wide range of modes. When set to conduct exhaustive search of it's relevant search space, it produces guaranteed optimal parallel plans 2 to 20 times faster than a version of Graphplan enhanced with CSP speedup methods. Initial experiments with heuristically pruning this search space have demonstrated that, though sacrificing the optimality guarantee, PEGG produces plans as short as Graphplan in terms of steps over a variety of domains. We discuss the reasons for the effectiveness of this approach by placing it in perspective to IDA* search and Graphplan's algorithm.

Keywords: planning, search, constraint satisfaction, machine learning, heuristics

1 Introduction

The AI planning problem can be cast in variety of different forms, some of the most effective currently being state space search, satisfiability (SAT), and the constraint satisfaction problem (CSP) format. [6,3,5] These paradigms each bring to bear certain inherent strengths and suffer intrinsic weaknesses in tackling planning problems. Predictably each system boasts on problems or classes of problems that it excels at and just as predictably research exposes it's shortcomings in other classes. There is obvious appeal to the idea of synthesizing two or more to exploit strengths and mitigate the weaknesses of each. Several investigations along these lines have been conducted [4,13]. We introduce here perhaps a more radical and versatile approach for effectively combining two of the more powerful planning paradigms of interest in recent years: state space and CSP-based search. The resulting system is capable of finding parallel plans that are optimal (or near optimal) in the number of steps and yet does so in run times comparable to some of the fastest planners currently in use.

State space search (SS), such as is employed by the HSP-r [6] and AltAlt [8] planners, exploits the view afforded by considering tiered sets of subgoals representing states. Theses planners greedily select the 'best' state to expand based on heuristic values calculated for the subgoals and may move to a solution quickly. However since the most effective heuristics often are not admissible, the solutions obtained may not be optimal and the search strategy may not even be complete [8]. SS systems are also not effective at building parallel plans where steps can contain more than one action.

CSP-based planners can readily adopt a more global, non-directional view in their search. They are not constrained to working on specific subsets of the problem variables, the 'states', in a contiguous fashion but can leverage global information about all the variables and their domains to expedite the search for a solution. Typically CSP planners employ a variety of mature speedup techniques such as dependency directed backtracking (DDB), explanation based learning (EBL), variable and value ordering, and "sticky values" [10]. However, these planners cannot readily leverage the state space view of the subsets of the problem variables that constitute 'states'. And unlike most BSS systems, larger problems can cause CSP formulations to confront working memory constraints.

The PEGG planner described herein combines both the CSP and state space views of planning, alternating CSP-style search facilitated by proven speedup techniques with a state space view that exploits heuristics to select the most promising set of subgoals to satisfy. The CSP-style search is rooted in Graphplan [1], which can be viewed as a system for solving "Dynamic CSPs" (DCSP) [12], and as such, finds parallel plans guaranteed to be the shortest in terms of number of steps. Graphplan carries out it's search for a problem solution by interleaving two distinct phases: a forward phase that builds a tiered "plan graph" structure followed by a phase that conducts backward search on that structure. PEGG captures the inter-level 'states' generated during the first Graphplan search phase in a concise structure we call the 'pilot explanation' (PE). Subsequent search phases are then conducted based on the PE. This pilot explanation affords us a state space view of Graphplan's CSP-style search process and facilitates a wide variety of ways of combining these alternate views of search. We will show that PEGG (Pilot Explanation Guided Graphplan) can leverage the strengths of each formulation without being crippled by their weaknesses. A version of PEGG (PEGG-h1) that employs a state space heuristic finds guaranteed optimal plans for a variety of problems well beyond Graphplan's reach. PEGG-h1 demonstrates speedups of 20x to over 400x over standard Graphplan and 1.5x to 20x compared to an enhanced version of Graphplan which was outfitted with all PEGG search and graph construction efficiencies that could be ported to it. Furthermore, PEGG-h1 tends to return step-optimal plans with fewer actions than Graphplan. Minor modifications gives us a version of PEGG that sacrifices the *guarantee* of optimality but produces very high quality plans nonetheless at speeds comparable to state-of-the art heuristic state space planners.

The paper is organized as follows. Section 2 discusses the motivation for building, maintaining, and using a pilot explanation structure to expedite search of Graphplan's plan graph.. Section 3 describes the major modifications made that enable PEGG to overcome the memory-bounded limitations of EGBG [7], the first system to use a PE. Section 4 describes the manner in which the PE allows PEGG to directly employ 'state space' oriented distance heuristics and gives results for two of the possible instantiations of this hybrid planning system. Section 5 discusses the relationship of this approach to well known search methods and related work. Section 7 presents our conclusions.

2. Exploiting Plan Graph Redundancy Using a Pilot Explanation

Due to its high visibility in the planning community since its introduction by Blum and Furst in 1995 [1], we do not describe here the details of the Graphplan algorithm. As a DCSP solver, Graphplan exhibits particular strengths that make it an attractive approach for solving planning problems:

- It finds parallel plans if possible (i.e. plans in which actions may appear in parallel). It is *guaranteed* to find a plan if one exists and it's *guaranteed* to be optimal in terms of number of steps
- In serial domains Graphplan returns optimal plans in terms of number of actions
- The plan graph captures both the relevant actions at each level and a measure of inconsistent actions (mutexes). This focuses search in a sound and complete manner during plan construction.
- Graphplan search naturally captures (in a lazy yet focused fashion) many inconsistent or unsatisfiable states in the form of memos. These constitute n-ary mutexes amongst propositions.
- By posing the planning problem as a CSP, it exploits the internal structure of a state such that it effectively handles situations with heavy interaction between actions and subgoals. Graphplan can also avail itself of the arsenal of CSP-based speedup techniques.

The motivation for the first system that employed the 'pilot explanation' concept [7] was simply to speed up Graphplan's iterative search process by avoiding certain redundancies and leveraging particular symmetries inherent to the plan graph and the search process itself. PEGG, the successor to EGBG still bases its search on the plan graph structure, but it exploits the PE concept in a number of interesting ways to give it great flexibility in conducting either state space or CSP-style search as appropriate in generating step-optimal solutions to planning problems. The following symmetrical or redundant features of the plan graph inspired development of the pilot explanation used by EGBG to expedite search at each new level:

1. The set of actions that can establish a given proposition at level k is a subset of the valid establishing actions for the proposition at level k+1.
2. The proposition goal set that is to be satisfied at a level k is exactly the same set that will be searched on at level k+1 when the plan graph is extended. That is, once the problem goal set is present at level k it will be present at all future levels.
3. For a given set of actions and propositions, the "constraints" (mutexes) that are active at level k+1 is a subset of the mutexes active at level k. (If A1 is initially non-mutex with A2 at level k it will never become mutex at higher levels.)
4. 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

We will refer to a particular backward search phase on a given length plan graph as a 'search episode'. The search conducted for the final search phase in which a solution is found we call the 'solution episode'. The above four factors, taken together, are responsible for considerable similarity (i.e. redundancy) in the search Graphplan's search for successive search episodes as the plan graph is extended. In fact, the backward search conducted at level k + 1 of the graph is essentially a replay of the search conducted at the previous level k with certain well-defined extensions. The original impetus behind constructing a pilot explanation during a search episode that fails to find a solution was to focus search in the next episode on areas of the search space that had not yet been explored.

Consider Figure 1's depiction of a Graphplan backward search episode in which a plan is sought that satisfies goals XYZ starting from plan graph level k. We've provided an abstract view of the search in terms of the various subgoal sets that become the target for further regression search at plan graph level n-1 once the subgoals at level n have been satisfied. The zoom view of the XWQ subgoal set reveals the CSP-style

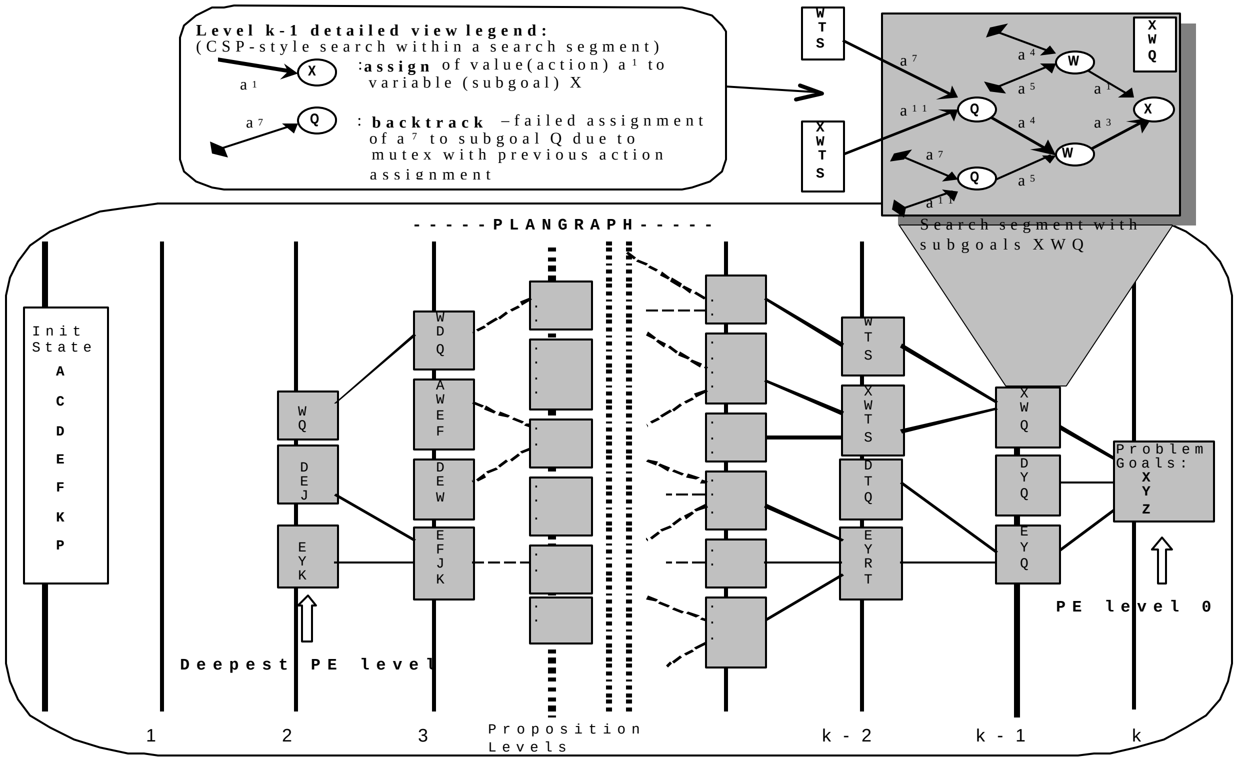
Level k-1 detailed view legend:
(CSP-style search within a search segment)
:assign of value(action) a1 to variable (subgoal) X
: backtrack –failed assignment of a7 to subgoal Q due to mutex with previous action assignment
Search segment with subgoals XWQ
- - - - - PLANGRAPH - - - - -
Init State A C D E F K P
Problem Goals: X Y Z
PE level 0
Deepest PE level
Proposition Levels
1
2
3
k - 2
k - 1
k

Figure 1. Pilot Explanation (PE) after search on problem goals X, Y, Z at plan graph level k.

search that Graphplan conducts at level k-1 in attempting to satisfy those goals. (Note that it finds two different sets of non-mutex actions satisfying XWQ, thus regressing to the WTS and XWTS subgoal sets.) Relative to this figure we informally define the following:

Search segment: a set of plan graph level-specific subgoals (i.e. a state) generated in regression search from the goal state (which is itself the first search segment) and tiered according to the level-by-level regression. Each level k search segment S_n contains a pointer to the parent search segment (S_p), that is, the state at level k+1 that gave rise to S_n after successfully assigning actions establishing all goals in that state. Thus a search segment represents a state plus some path information but we may use the terms interchangeably. In Figure 1, XWQ is one such search segment.

Pilot explanation (PE): the entire linked set of search segments representing the search space visited in a Graphplan backward search episode. It's convenient to visualize it as in Figure 1: 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.

Armed with these simple definitions and the four plan graph symmetry features described above we can make these key observations:

- The PE, after a search episode 'n' on plan graph level k is a *lower bound* on the set of states that will be visited when backward search is conducted in episode n+1 at level k+1. (This bound can be visualized by sliding the fixed tree of search segments in Figure 1 up one level.)
- *Every* set of subgoals (search segment) reached in the backward search of episode n, starting at level k, will be generated again by Graphplan in episode n+1 starting at level k+1.
- Any *new* search segment that is generated in episode n+1 must be a descendant of at least one of the search segments already in the PE after episode n.
- Consider segment XWQ generated at level k-1 in search episode n (Figure 1 detailed view). When Graphplan conducts episode n+1 search on the problem goals at level k+1 this same search segment (subgoal set) will occur at level k. There are only two conditions under which the episode n+1 search trace for segment XWQ will differ from the episode n search trace:
    1. A new action appears for the 1st time in level k that gives X,W, or Q
    2. Two actions giving at least two of the XWQ goals were (dynamic) mutex at level k-1 and become non-mutex at level k.
- As long as the backward search algorithm memoizes a search segment's goals when they cannot be consistently satisfied, no two search segments for a given plan graph level will have the same goals, and each search segment will have only one parent.

Based on these observations we developed a Graphplan-based planner that builds a pilot explanation during the first backward search episode, and then uses it to focus search in all subsequent episodes, extending the PE as warranted. In each successive episode, after the plan graph is extended, the PE -a concise trace of the previous episode's search space- is *translated* up one level and the backward search for a plan is conducted by *visiting* the search segments in the PE in some order. Here we define some processes:

PE translation: For a search segment in the PE associated with plan graph level j after search episode n, associate it with plan graph level j+1 for episode n+1. Iterate over all segments in the PE.

Visiting a search segment: For segment S_p at plan graph level j+1, visitation is a 3 –step process:

1. Perform a memo check to ensure the subgoals of S_p are not a nogood at level j+1
2. Initiate Graphplan's CSP-style search to satisfy the segment subgoals beginning at level j+1. A child search segment is created and linked to S_p (extending the PE) whenever S_p's goals are successfully assigned entailing a new set of subgoals to be satisfied at level j.
3. Memoize Sp's goals at level j+1 if all attempts to consistently assign them fail.

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. The search process for the resulting planner essentially alternates selection of a promising state to visit with dynamic CSP-type search on the state's subgoals. If the DCSP search fails to find a plan, the planner selects another PE search segment to visit.

An important facet of this description of PE-based planning is that it *leaves open the strategy to be used in visiting the search space* represented by the PE. We expand on this in Section 4, but we note here a somewhat subtle issue associated with the order of search segment visitation. If a parent segment at level k is visited *prior* to one of its level k-1 children (say S_c), the subgoals of S_c will be generated again and a redundant search segment added to the PE as search at level k-1 proceeds. One means of efficiently avoiding this inflationary impact on the PE is to always visit PE search segments in a 'bottom up', level-by-level fashion. In this way the subgoals of a parent's child segments will have been memoized prior to the Graphplan-style search on the parent's goals, and creation of redundant segments will be blocked by memo checking. This was the default strategy of EGBG, the first PE-based planner.

EGBG minimized the redundant search performed *within* a search segment each time it is visited across episodes. By storing enough of the intra-segment search trace in a search segment it avoided most of the redundant mutex-checking and action assignment each time the segment is revisited. EGBG demonstrated speedups of 2 – 20x over Graphplan on a variety of modest sized problems [7]. However, as discussed there, the combined storage requirements of the multi-level plan graph and the pilot explanation pushed larger problems out of reach for this Lisp-based implementation. Our current PE-guided system, PEGG, not only solves the memory bound issues, but greatly reduces planning time and extends the system's reach by exploiting a variety of techniques made possible by the PE. We discuss these in the next section.

3 Addressing memory limitations *and* accelerating search

Our work with pilot explanation-based planning has shown how closely related the reduction in memory demands is with accelerated planning for such systems. A discussion of all the advances relative to EGBG that led to the PEGG system is beyond the scope of this paper. However we briefly describe here five major techniques or that were implemented either to reduce memory constraints *or* to speedup planning, and we note that three of them contribute significantly to both goals.

Domain preprocessing and invariant analysis: The speedups attainable through preprocessing of domain and problem specifications are well documented [9,16]. For planners that employ a plan graph these approaches also reduce memory requirements to the extent that they identify propositions that do not need to be explicitly represented in each level of the graph. For planners that employ a pilot explanation this benefit is compounded since it directly impacts the number of subgoals that must be saved in each search segment. PEGG employs domain preprocessing that is related to but not as extensive as the processing done by the TIM subsystem used by STAN [9].

Bi-level plan graph: It's been known for some time that the original multi-level plan graph structure employed by Graphplan could be represented as an indexed two level structure [9]. For PEGG this has a three-fold benefit; it reduces the graph's memory demands, facilitates faster building/extension of the plan graph, and reduces PE translation to simply incrementing each search segment's graph level index.

Explanation Based Learning and Dependency Directed Backtracking: Conditions under which EBL and DDB can accelerate the search process have been well documented [10,11]. Less obvious perhaps is the role it 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. The PE grows in direct proportion to the search space actually visited, so such techniques that prune search provide yet another benefit in the form of reduced memory demands for a system such as PEGG.

Value and Variable Ordering: Value and variable ordering are also well known speedup methods for CSP solvers. The impact of variable and value ordering on Graphplan performance was examined in [8] where the plan graph was exploited to extract level-based heuristics for ordering subgoals ('variables') and goal establishers ('values'). They demonstrated modest speedups for Graphplan under some of the heuristics, though it varied across domains/problems. We found that using the plan graph based "level" value for both goal and action ordering provided significant speedup in a variety of problems while incurring very little overhead. This heuristic assigns to each action and proposition the plangraph level at which it first appears. We have used it in all PEGG results reported here.

A 'skeletal' pilot explanation: EGBG avoided redundant mutex checking to a maximal extent by storing information about the results of search *within* each search segment. However, examining EGBG's performance reveals that the lion's share of the credit for its speedup relative to Graphplan seldom goes to it's ability to avoid the redundant action and proposition mutex checking across successive search episodes. Rather it springs from the default order in which EGBG processes the search segments in the PE trace. Processing the PE search segments in a 'bottom up' fashion is a logical 'heuristic' to employ since this section of the search space lies closest to the initial state (in terms of plan steps). This simple approach by itself can have a major impact on search time in the final search episode where most of the computation time is often logged.

Consider search conducted using the PE in the solution episode of a problem. Suppose we visit search segments in a bottom up manner and visit one, S_{ss}, in the lower levels that can be extended to the initial state, thereby finding a solution. We will have completely avoided all the higher level search required to reach S_{ss} from the top level problem goals. 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.* We now make a somewhat curious but useful empirical observation:

> For *all* planning problems analyzed to date using a PE-based planner, a seed segment has been found within the lowest 4 levels of the PE in the solution episode. In other words, when a solution can be extracted from a given plan graph, EGBG or PEGG have only needed to visit the search segments in the lowest 4 levels of the PE. (In fact, for the great majority of problems there's a seed segment in the lowest 2 PE levels.)

This suggests a number of search strategies that we will touch on later. It also motivates the move to a skeletal PE in which we drastically reduce memory requirements by no longer saving the intra-segment CSP search trace. PEGG, unlike EGBG, retains only the snapshot of the search space states visited in an episode, at the expense of performing Graphplan's redundant action assignment mutex checks within each segment. As we show in the next section, this is a good tradeoff. PEGG uses the PE to adopt a state space view in choosing the best search segments to visit and alternates with Graphplan's CSP view to quickly extend the optimal plan tail using powerful CSP speedup methods.

Table 1 provides some runtime comparisons of standard Graphplan with two versions that have been augmented with combinations of the techniques discussed above. Seven of the 12 problems can only be solved by Graphplan when the enhancements have been added. For the 5 problems that standard Graph-

Problem	**Standard GP** cpu sec	**GP w/ bi-level PG & domain preprocessing** cpu sec	**GP w/ bi-level PG, dom. preprocessing, EBL/DDB, goal & action ordering** cpu sec	**PEGG** cpu sec	**SPEEDUP** (PEGG vs. column 4 GP)
bw-large-B (18/18)	234.0	219.3	101.0	22.3	4.5x
Rocket-ext-a (7/36)	1131	127.9	39.8	3.0	13.3x
Rocket-ext-b (7/36)	846	99.0	27.6	4.9	5.6x
att-log-a (11/79)	~	728.3	31.8	2.8	11.3x
Gripper-8 (15/23)	~	83.2	28.8	17.6	1.6x
Tower-7 (127/127)	~	94.7	114.8	15.0	7.7x
Tower-9 (511/511)	~	~	~	118	>1000x
Mprime-1 (4/6)	738	15.0	4.8	4.9	(1.02x)
8puzzle-1 (31/31)	~	~	95.2	64.0	1.5x
8puzzle-2 (30/30)	~	~	87.5	41.8	2.1x
8puzzle-3 (20/20)	~	84.3	19.7	3.0	6.6x
TSP-12 (12/12)	~	1020	131	10.7	12.2x

Table 1. Comparisons of Graphplan, enhanced Graphplan and PEGG.
Times are for a Pentium 500 mhz machine, Linux, 512 M RAM, Allegro Lisp, (excluding gc time)
Parentheses after problem give Graphplan's optimal # of steps/ # of actions. (All systems in table find equivalent optimal plans) Stnd Graphplan based on Peot & Smith's Lisp implementation.
Goal & action ordering based on plangraph 'level' distance. ~ :means no solution was found in 30 minutes.

plan *can* solve the speedups exhibited by the fully enhanced version range from ~2x to 154x. Clearly these enhanced versions significantly raise the performance bar for any comparison with PEGG. The last two columns provide this comparison and show that PEGG speedups range from essentially 0 to over 1000x, with a median for this sampling of problems of about 7x.

Stepping back for a moment, it's worth considering the class(es) of problems for which the pilot explanation will actually accelerate the solution search. In problems for which a solution can be extracted from the plan graph at the *first* level in which the goals all appear and are non-mutex, there is only one search episode. As such, the PE that is built never gets used and the cost of building it serves only to add to the solution time (though empirically PEGG has run only 5 – 15% slower for such problems). The Table 1 problems for which PEGG has a decided advantage are multi-episode problems in which the primary benefit afforded by the PE accrues in the final search episode. And this advantage is based primarily in the 'heuristic' of visiting search segments in the lowest PE levels first. Since the PE also provides us with a concise state space view of PEGG's search space we consider in the next section how the 'distance based' heuristics employed by state space planners such as UNPOP [17], HSP-R [6], and AltAlt [8] might be put to work by PEGG to further speed up search in problems with multiple search episodes.

4. Exploiting the state space view

We report here on two approaches to applying distance based heuristics to PEGG's search: 1) Ordering all search segments in the PE according to a given state space heuristic and visiting all of them in order 2) Ordering all search segments in the PE according to a given state space heuristic and visiting a subset of them in order. The first approach maintains Graphplan's guarantee of step optimality but focuses significant speedup only in the final search episode. The second approach sacrifices the guarantee of optimality in favor of pruning search in *all* search episodes. Still, we shall see that there's a high likelihood that optimal plans are found in spite of the sacrificed guarantee.

As soon as we abandon the strategy of visiting the PE segments in a bottom-up fashion, we run the risk of producing an inflated, redundant pilot explanation, as discussed at the end of Section 2. We investigated several means of handling this issue, but the details are beyond the scope of this paper. The approach used by versions of PEGG reported here is to simply allow duplicate segments to be generated and inserted, and then to prune the *original* child segments when they are visited later in the episode. These original segments can be identified via the memoizing conducted as part of the visitation process, as defined in Section 2.

The heuristic f-value for a state (search segment) S_n is defined as:

$\mathbf{f = g + w * h}$

where g is the 'cost' of reaching S_n *from the problem goals (# of steps* or *actions for Graphplan)*
h is an estimate of the distance to the initial state
w is a weight parameter

Note that the g value has two possible definitions here. If we define it as the number of steps, the g-value for a search segment is just the PE level number. In this case any cost associated with parallel actions within a step is ignored and the range of possible g values is just the number of levels in the PE. On the other hand if g is defined in terms of number of (non-persist) actions we bias the search towards finding a plan with the fewest number of actions. The range of possible g values will in general be much greater. Thus given two search segments (states) at level k with the same h values, the planner would choose to visit the one entailing the fewest number of actions, parallel or otherwise. So setting g to the number of non-persists actions effectively biases PEGG towards returning a step-optimal plan with the fewest number of actions, and we use this setting for all heuristic-guided PEGG results reported here. Without actually conducting a branch and bound search for the least cost plan in a solution episode, we cannot guarantee we've found the step-optimal plan with fewest actions, of course. We intend to report on work along these lines elsewhere.

The h-values we consider here are taken from the distance heuristics described in [8]. The authors discuss the tension between using an admissible heuristic that ensures an optimal solution vs. a more informed inadmissible heuristic that may greatly speedup the BSS planner at the expense of loosing optimality (or even completeness). As long as PEGG visits the entire PE in each search episode, it can employ an inadmissible heuristic to direct it's search space traversal without losing it's guarantee of finding a step-optimal plan. We've restricted this analysis to a single heuristic that the authors called 'adjusted-sum' [8]. It is inexpensive to calculate for a plan graph based planner and was found to be quite effective for the BSS planners they tested. A brief overview of the basis of the heuristic is provided here and the reader is referred to reference 8 for details.

The heuristic cost *h(p)* of a single proposition is computed iteratively to fixed point as follows. Each proposition *p* is assigned cost 0 if it's in the initial state and ∞ otherwise. For each action, *a*, that adds *p*, *h(p)* is updated as: $h(p) := min\{h(p), 1+h(Prec(a)\}$ where *h(Prec(a))* is computed as the sum of the h values for the preconditions of action a. 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 now be stated:

$$h_{adjsum}(S) \coloneqq \sum_{p_i \in S} \cos t(p_i) + lev(S) - \max_{p_i \in S} lev(p_i)$$

This is essentially a 2-part heuristic. The summation estimates the cost of achieving *S* under the assumption that the cost of achieving the propositions in a state are independent. The last two terms provide a measure of the additional cost incurred by negative interactions in achieving the state's propositions.

Returning to the expression for *f* above, we set the value of 'w' to 5 based on empirical studies conducted with state space planners [8]. It may well be that there are more effective w values for the sort of k-length plan search conducted by PEGG, but we leave that as an open issue for now.

4.1 Complete search with distance heuristics

In this instantiation of PEGG, which we'll refer to as PEGG-h1, we retain the guarantee of step-optimality by visiting all search segments in the PE in each search episode. The adjusted-sum heuristic value is calculated for each segment in the PE and the entire corpus of segments is then sorted with lowest f-value first. Figure 2 provides a high level view of the planning process for the heuristic version of PEGG[1]. Note that the state space distance heuristic is used only to direct the traversal of the search space represented by the search segments in the pilot explanation. It's feasible to also employ the heuristic in the Graphplan-style search that ensues when a segment is visited, but we leave that discussion to Section 8, Future Directions. Anticipated speedups are based on the hope that in the *solution episode* the heuristic will direct PEGG to a seed segment from the PE sooner than the blind, level-by-level visitation scheme. We'd expect the biggest payoff in problems such as "bw-large-b" of Table 1, where the lowest level seed segment in the PE lies four levels above the deepest PE level when the solution episode commences. For that problem, there are some 1500 search segments that the level-by-level PEGG version has to visit before it gets to the seed segment that extends to a plan.

The results reported in Table 2 bear out this prediction. In columns 3, 4 and 5 are reported runtimes over a variety of problems for three planners: the fully enhanced version of Graphplan described in Table 1, col-

[1] The optional checks regarding the 'heuristic threshold' apply only to the second approach discussed in this section: using heuristics to also shortcut search in non-solution bearing episodes.

Figure 2. PEGG pseudo code (state space heuristic version)

```
1.  Build plan graph to the 1st level having non-mutex goals
2.  Conduct the 1st Graphplan style backward search episode using CSP speedup techniques.
    In so doing, build the initial PE .
3.  IF a solution is reached: DONE
    ELSE extend plangraph and use the PE to direct the subsequent search episode as follows:
     A.  Enqueue search segments from the entire PE according to a given state space heuristic
     B. IF f-value of the front (best) segment is below (optional) heuristic threshold visit the segment:
             * Perform a memo check on the segment subgoals. If they constitute a nogood remove the
             segment from the PE, advance to next segment, return to B.
             * Conduct Graphplan-style search on segment subgoals using CSP speedup techniques.
               Extend PE structure in the process
             IF a solution is reached: DONE
               ELSE memoize the segment subgoals, advance to next segment and return to B.
           ELSE (optional) heuristic threshold has been exceeded, perform memo check on remaining
               segments in queue, removing any with nogood goals.
     C. Extend the plan graph one level & Translate the PE up one level
     D. Return to A
```

umn 4 (which we call GP-E), PEGG with the same enhancements as GR-E, and PEGG-h1 which has these same enhancements *plus* state space search guided by the “adjusted-sum” heuristic.

Examining the actual PE search trace for these problems reveals that the advantage of PEGG-h1 over PEGG is indeed expressed clearly in problems for which the lowest seed segment lies ‘above’ the lowest PE levels. Table 2 problems in this category include bw-large-b, rocket-ext-b, gripper-15, and eight-puzzle-1. The state space heuristic successfully identifies the seed segments, which lie in higher PE levels for these problems, and orders them in front of many other lower lying segments that PEGG winds up visiting first. PEGG-h1, on average, cuts the solution time in half for these problems while maintaining the guarantee of returning a step-optimal plan.

Problem	**Stnd GP** cpu sec	**GP-E** (enhanced Graphplan) cpu sec (steps/acts)	**PEGG** cpu sec (steps/acts)	**PEGG-h1** heuristic: adjsum cpu sec (steps/acts)	**PEGG-h2** heuristic: adjsum cpu sec (steps/acts)	**Alt Alt** (Lisp version) cpu sec (/ acts) **heuristics:** **adjusum2**	**combo**
bw-large-B	234.0	101.0 (18/18)	22.3 (18/18)	12.2 (18/18)	9.4 (18/18)	87.1 (/ 18)	20.5 (/28)
bw-large-C	1131	~	~	~	93.9 (28/28)	738 (/ 28)	114.9 (/38)
Rocket-ext-a	846	39.8 (7/36)	3.0 (7/36)	2.8 (7/34)	2.1 (7/34)	43.6 (/ 40)	1.26 (/ 34)
Rocket-ext-b	~	27.6 (7/36)	5.5 (7/36)	2.7 (7/36)	2.7 (7/34)	555 (/ 36)	1.65 (/34)
att-log-a	~	31.8 (11/79)	2.8 (11/79)	2.8 (11/72)	2.2 (11/62)	36.7 (/56)	2.27(/ 64)
att-log-c	~	~	~	~	41.0 (/66)	53.3 (/ 61)	3.58 (/70)
Gripper-8	~	28.8 (15/23)	17.6 (15/23)	16.6 (15/23)	8.0 (15/23)	14.1 (/ 45)	*
Gripper-15	738	~	~	47.5 (36/45)	16.7 (36/45)	14.1 (/ 45)	*
Tower-7	~	114.8 (127/127)	15.0 (127/127)	14.3 (127/127)	9.9 (127/127)	*	
Tower-9	~	~	118 (511/511)	91 (511/511)	42.6 (511/511)	*	48.5
Mprime-1	~	4.8 (4/6)	4.9 (4/6)	3.6 (4/6)	2.9 (4/6)	*	
8puzzle-1	~	95.2 (31/31)	64.0 (31/31)	39.1 (31/31)	17.2 (31/31)	143.7 (/ 31)	20.2 (/?)
8puzzle-2		87.5 (30/30)	41.8 (30/30)	31.3 (30/30)	11.0 (30/30)	348.3 (/ 30)	7.42 (/ ?)
8puzzle-3		19.7 (20/20)	3.0 (20/20)	2.7 (20/20)	1.8 (20/20)	62.6 (/ 20)	11.0 (/ ?)
grid4		~ (?/?)	~	75 (18/18)	75 (18/18)	30.5 (/ 18)	14.5 (/ 18)

Table 2 PEGG heuristic guided versions vs. enhanced Graphplan and a BSS heuristic planner
GP-E: Graphplan enhanced with bi-level PG, domain preprocessing, EBL/DDB, goal & action ordering
PEGG: Same enhancements as GP-E
PEGG-h1: complete PE search, guided by adjsum state space heuristic
PEGG-h2: abbreviated PE search, guided by adjsum state space heuristic
Parentheses next to cpu time give # of steps/ # of actions in solution
All planners in Allegro Lisp, runtimes (excl. gc time) on Pentium 500 mhz, Linux, 512 M RAM
“adjusum2” :AltAlt run under the “adusted sum2” heuristic
“combo” :AltAlt run under “combination” heuristic, run using fast plan graph construction routine.
~ indicates no solution was found in 30 minutes * indicates problem wasn’t run

The impact of this solution episode heuristic ordering is even more pronounced with gripper-15 and the gridworld problem, grid4. Easily bested by PEGG-h1, PEGG cannot solve either of these in 30 minutes, even though both planners take roughly the same length of time to get to the search episode in which a solution can be found. Note also, that Graphplan guarantee of step-optimality does *not* extend to finding plans with the fewest numbers of actions. As such, it's not surprising to find that PEGG-h1, with it's heuristic g-value based on the number of actions, finds step-optimal plans with *fewer* actions than that found by Graphplan for two problems (rocket-ext-a, att-log-a).

4.2 Abbreviated search with distance heuristics

With an instantiation of PEGG we call PEGG-h2, we seek an effective alternative to Graphplan's search strategy of exhausting 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 characteristic to Graphplan's solutions, but for many sizeable problems it exacts a high cost to ensure what is, after all, only one aspect of plan quality. In the years since it was developed (1995) many different genres of planners have successfully tackled problems beyond Graphplan's reach [3,4,5,6], and some have demonstrated they can generate plans of comparable quality (albeit without any 'guarantees'). PEGG-h2 leverages the state space view afforded it by the PE along with a suitable heuristic, to visit only the most promising search segments in each search episode.

Consider again the algorithm of Figure 2. For PEGG-h2, exhaustive search of the PE search space is truncated in substep B with the heuristic 'threshold test'. The intent is to avoid visiting search segments that hold little promise according to the heuristic used. When the first segment exceeding this threshold is reached on the sorted queue the search episode ends. A memo check is performed on all the remaining segments on the queue (to identify redundant segments for pruning) but *no search* is conducted on them, nor are their subgoals memoized. If the ranking heuristic is one that can effect a *decrease* in a state's f-value as it translates to a higher plan graph level (the adjusted-sum is one such heuristic) then these unpromising segments are retained in the PE to be considered in the next episode. Otherwise they could logically be deleted since they will never pass the threshold test.

The problem of devising a highly effective threshold test is an interesting one that must reconcile the competing goals of reducing search in non-solution bearing levels to a minimum while maximizing the likelihood that *a plan segment* is visited in the solution episode. The narrower the search window, the more pressure is put on the state space heuristic that ranks the segments to be visited. But there are two observations that make the heuristic task less daunting: 1) PEGG-h2 will return a step-optimal plan as long as, at some point in the first solution-bearing episode, its search strategy leads it to visit *any* plan segment in *any* plan latent in the PE (including the single top segment in the PE). 2) There are *many* step-optimal plans latent at the solution-bearing level for most problems. We again defer to later work an investigation of the most effective sort of search segment visitation threshold test and report here the results of using a simple 2-phase test:

- If at the start of a search episode there are less than N in the PE, visit all search segments.
- For episodes in which the PE has >N segments, visit the top M% of them, based on f-value

With a few trial runs to characterize a sampling of problems, we set N=100 and M= 50% and report the results in Table 2. We also report there the performance of AltAlt, one of the faster BSS heuristic planners, on the same problems. The last column gives the runtime and plan 'quality' in terms of number of actions for what proved to be two of the best heuristics for AltAlt overall [8]. Like virtually all state space heuristic planners, AltAlt does not find possibly concurrent actions so as to build parallel plans, so the number of steps in its plans is not reported.

PEGG-h2 (N:100, M:50%) again extends the size of problem that can be handled. Both bw-large-c and att-log-c could not be solved by previous versions of PEGG or Graphplan. PEGG-h2 solves all problems faster than PEGG-h1 (by an average of ~2x) and compares nicely to AltAlt, even bettering the latter's best heuristic in 7 of 13 problems for which data were available. More importantly, we see convincing evidence that PEGG-h2 can not only return step optimal plans, but it can also find shorter ones in terms of number of actions. In every case it returned a plan of the same step-length as Graphplan (or complete PEGG) and for 3 problems it found shorter plans in terms of actions.

5. Discussion and Related Work

The connection between Graphplan's search and solving a "dynamic constraint satisfaction problem" (DCSP) was made by Kambhampati [13]. Bonet and Geffner [6] were among the first to note that Graphplan search can be viewed also as state space search, arguing that overall process is a version of IDA* search. These two viewpoints actually look with different granularities at the search process. A single Graphplan backward search episode is described most completely as a DCSP while Graphplan's iterative character in solving a problem requiring multiple episodes is akin to IDA* search. *Both* views help in understanding PEGG's effectiveness, so we reference aspects from both here in order to clarify the role of the pilot explanation in expediting search.

Graphplan solves a planning problem in the direct DCSP fashion described by Mittal & Falkenhainer [15]. Starting with the problem goals as 'active variables' it seeks a satisfying assignment for them subject to the variable domains and the mutex constraints of the highest plan graph level. Taken by itself, this sub-problem is solved as any CSP; it is non-directional in nature and can take advantage of a variety of CSP speedup techniques. We'll refer to the assignment search for one such set of propositions as an epoch. If successful, the assignments may activate new variables which become the subgoal propositions for the next epoch.

Clearly the set of subgoals (active variables) to be assigned in any epoch can be viewed as a state. As such, DCSP can be seen as occupying a middle ground between CSP search and state space search. It decomposes the overall problem into a series of sub-problems, each analogous to a state, and then conducts non-directional CSP-style search on each sub-problem individually.

The Graphplan connection to IDA* [6], is based on making two mappings 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 each Graphplan node is a 'state' comprised of the subgoals that result from its regression search.
2. The upper bound that is 'iteratively deepened' ala IDA* is the heuristic f-value for node-states, where:
   g :the cost of reaching the node-state from the goal state in terms of number of DCSP epochs (simply the difference between the number of the highest plan graph level and the level number at which the node-state subgoals are to be assigned).
   h :the distance to the initial state in terms of DCSP epochs (or again, associated plan graph levels)

   The iterative deepening in Graphplan is done implicitly each time the plan graph is extended for another search episode.

From the IDA* perspective on Graphplan's search, PEGG directly addresses two important shortcomings. First, it's long been recognized that IDA*'s difficulties in some problem spaces can be traced to using *too little* memory. The only information carried over from one iteration to the next is the upper bound on the f-value. Graphplan partially addresses this with its memo caches that store learned nogoods for use in successive episodes. The pilot explanation goes 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 [17] and SMA* [16] which lie in the middle ground between the memory intensive A* and IDA*. We have not yet been forced to limit the size of the

PE in the manner that SMA* prunes its search queue, but it is a straightforward matter to do so should the demands of a problem dictate.

The second shortcoming of the IDA* nature of Graphplan's search becomes apparent upon consideration of the f-value as defined for an episode (mapping #2 above). All node-states generated in a given Graphplan episode have *the same f-value,* so that within an iteration (search episode) there's no discernible preference for visiting one state over another. Since the PE retains a view of the previous episode's state space, PEGG can apply any of the powerful state space heuristics (those derived from the plan graph are obvious choices), effectively layering a 2nd-tier of heuristic control onto Graphplan's search. This means that PEGG is not as constrained by the depth-first search nature of Graphplan's backward search, and can move about the state space visited in the previous episode according to the assessed desirability of each state. Note that even though we might employ what are generally more powerful *inadmissible* heuristics in traversing the PE search space, we can retain the guarantee of step-optimality, if we choose, as long as we enforce Graphplan's encoded admissible heuristic by visiting all the search segments in the PE. Haslum and Geffner [18] describe an entirely state space based planner that also finds optimal parallel plans by layering an inadmissible node selection heuristic on top of an IDA* search algorithm using an admissible heuristic. We hope to compare performance on similar problems in future work.

PEGG is the first system 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. Kautz and Selman [14], alternately adopt CSP and SAT views in their Blackbox system, converting a k-level plan graph into a SAT encoding whereupon a k-step solution is sought. Do and Kambhampati [4], similarly alternate between extending a plan graph and converting it, but they transform the structure into CSP format and search to satisfy the constraint set in each search phase.

6. Conclusions and Future Work

The planning paradigms developed to date each bring to bear specific inherent strengths and suffer intrinsic weaknesses in tackling problems in the field. We have introduced a flexible approach for melding two of the most effective views of planning, CSP and state space search, and demonstrated that it can exploit strengths of both. Graphplan's DCSP approach to planning remains perhaps the most robust and efficient method for finding parallel, optimal plans. Various studies have demonstrated that heuristic state space search is currently amongst the fastest approaches for generating 'good quality' plans for some of the largest benchmark problems. The PEGG system captures in its pilot explanation, a state space view of its underlying Graphplan, CSP-style search. This enables it to interleave search directed by plan graph based heuristics with DCSP extension of partial plans implicit in the PE. By varying the portion of the PE search space that is visited in each search episode the system can exhibit performance ranging from returning guaranteed optimal plans two or more orders of magnitude faster than Graphplan, to performance comparable to state-of-the-art state space planners while returning very close to optimal quality plans.

PEGG provides a variety of control points for tuning and adapting its performance, including application of a variety of heuristics and controls on how much of the PE search space can be pruned without significantly degrading the quality of the plans being returned. We intend to further investigate possible domain and/or problem-specific aspects of such search control and the extent to which they might be automatically assessed or learned online. Another open question is whether the state space heuristics, in addition to directing traversal of the PE, can be exploited in the DCSP phases of PEGG's problem solving process. We are also interested in a longer and wider few aimed at extending this approach to handling resource based and temporal planning problems.

References

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

[2] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *JAIR*, 10(1-2) 1999.

[3] H. Kautz and B. Selman. Pushing the envelope: Planning, prepositional logic and stochastic search. *In Proc. AAAI-96.*

[4] Minh Binh Do and Kambhampati, S. Solving Planning-Graph by compiling it into CSP. In *Proc. AIPS-2000.*

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

[6] Bonet, B. and Geffner, H. Planning as heuristic search: New results. In *Proc ECP-99,* 1999.

[7] Zimmerman, T. and Kambhampati, S. 1999. Exploiting Symmetry in the Planning-graph via Explanation-Guided Search. In *Proc AAAI-99,* 1999.

[8] Nguyen, X. and Kambhampati, S. Extracting effective and admissible state space heuristics from the planning graph. In *Proc. AAAI-2000.*

[9] Fox, M., Long, D. 1998. The automatic inference of state invariants in TIM. *JAIR* 1999.

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

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

[12] 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.

[13] Kambhampati, S., Parker, E., Lambrecht, E. 1997. Unerstanding and extending graphplan. In Proceedings of 4th European Conference on Planning.

[14] Kautz, H. and Selman, B. Unifying SAT-based and Graph-based Planning. In *Proc of IJCAI-99,* Vol 1, 1999.

[15] Mittal, S., Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proc. AAAI-90*

[16] Gerevini , A., Schubert, L. 1996. Accelerating Partial Order Planners: Some techniques for effective search control and pruning. JAIR 5:95-137.

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

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