Generating parallel plans satisfying multiple criteria in anytime fashion

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

Abstract

We approach the problem of finding plans based on multiple optimization criteria from what would seem an unlikely direction: find one valid plan as quickly as possible, then stream essentially *all* plans that improve on the current best plan, searching over incrementally longer length plans. This approach would be computationally prohibitive for most planners, but we describe how, by using a concise trace of the search space, the PEGG planning system can quickly generate most, if not all, plans on a given length planning graph. By augmenting PEGG with a branch and bound approach the system is able to stream parallel plans that come arbitrarily close to a user-specified preference criteria based on multiple factors. We demonstrate in preliminary experiments on cost-augmented logistics domains that the system can indeed find very high quality plans based on multiple criteria over reasonable runtimes. We also discuss directions towards extending the system such that it is not restricted to Graphplan's scheme of exhaustively searching for the shortest step-length plans first.

I. Introduction

From a classical planning perspective a basic, multiple criteria optimization problem might entail finding a plan that optimizes two factors:

x: the number of time steps
y: the total 'cost' of the plan

Here the optimization itself will be with respect to some user-specified criteria involving x and y. Graphplan is a well-known classical planner that, in spite of the more recent dominance of heuristic state-search planners, 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). However, there is no known practical approach for finding cost-optimal plans with Graphplan, let alone optimizing over some arbitrary weighting of time steps and cost. We describe and report on initial experiments with a Graphplan-based system that streams a sequence of plans that increasingly approach a user-specified optimization formula based on multiple criteria. This system, which we call Multi-PEGG, seeks to find the plan that comes closest to matching the user's preference expressed as a linear preference function on two variables. (e.g. $\alpha x + \beta y$, where x and y might be defined as above). As we'll discuss in Section V (future work) extending the system to handle more than two criteria is straightforward, as is implementation of criteria such as 'the least cost plan with no more than k steps'.

Consider first how a plan satisfying multiple criteria might be generated by Graphplan if computation time were not an issue. By alternating search episodes on the planning graph with extensions of the graph, Graphplan's algorithm is guaranteed to return the shortest plan in terms of time steps (where a step might include multiple actions that do not conflict). If Graphplan finds its shortest valid plan for the given problem on a k-level planning graph, a modest modification of the program could, in principal, find *all* possible valid k-length plans by conducting exhaustive search on the same planning graph.[1] The final set of plans could then be post-processed to find the best one in terms of any other optimization criteria giving us, for example, the least cost, k-length plan. However, not only is this approach computationally impractical for many problems/domains, but it can only handle a small subset of the multi-objective criteria one could envision. Such a system for example, could not satisfy a user request for the least-cost plan of any length.

In a naive attempt to extend the system capabilities so its scope includes plans of length greater than k, we might iteratively extend the planning graph, restarting the solution search for valid plans at each successive level. If we have a means of calculating 'cost' for the subgoal sets generated during the regression search, branch and bound techniques might be applied after finding the first valid plan to prune some of this search space. Nonetheless, this will clearly be an intractable approach for any problem of sufficient size to be of interest.

The PEGG (Pilot Explanation Guided Graphplan) planning system dramatically boosts Graphplan's ability to find step optimal plans by taking advantage of certain symmetries and redundancies in its search process [Zimmerman and Kambhampati, 2001, 2000]. We report here on preliminary work with extending PEGG in such a way that it leverages those planning graph related

[1] There are few subtleties involved in doing this. For example, care must be taken so that the subgoal sets generated in the regression search that directly leads to each valid plan are *not* memoized. The standard Graphplan goal assignment routine memoizes goal sets at each planning graph level as it backtracks.

symmetries to efficiently generate *all* plans of interest on any length graph. The 'Multi-PEGG' planner, which we focus on in this study, employs this capability together with a heuristic-based branch and bound strategy to generate a stream of increasingly higher quality plans (relative to the user's definition of quality). Given a variety of linear user preference formulas, we show that this approach can efficiently stream monotonically improving solutions for two different logistics domains augmented with action cost values.

The rest of this paper is organized as follows: Section II gives an overview of the PEGG system on which Multi-PEGG is based, and reports on its performance relative to Graphplan and one of the faster heuristic state space planners. Section III describes the extensions to PEGG that allow it to efficiently extract many, if not all, valid plans from a given length planning graph in reasonable time. Section IV then describes how Multi-PEGG exploits this capability along with branch and bound techniques to stream plans that come increasingly closer to a user-specified quality metric based on multiple criteria. Section V contains our conclusions and ideas for future work.

II. Using memory to expedite Graphplan's search for step-optimal plans

The approach we adopt to finding plans satisfying multiple criteria is rooted in the ability of the PEGG planner to efficiently find all valid plans implicit in a given length planning graph. The planning system makes efficient use of memory to transform the depth-first nature of Graphplan's search into an interactive state space view in which a variety of heuristics are used to traverse the search space [Zimmerman and Kambhampati, 2001, 2002]. It significantly improves the performance of Graphplan by employing available memory for two purposes: 1) to avoid some of the redundant search Graphplan conducts in consecutive iterations, 2) and (more importantly), to transform Graphplan's iterative deepening depth-first search into iterative expansion of a selected set of states that can be traversed in any desired order. We briefly review in this section the PEGG algorithm before describing how it can be adapted to find all plans on the graph.

The original motivation for the development of PEGG and the related planner that preceded it, EGBG [Zimmerman and Kambhampati, 1999], was the observation of redundancy in Graphplan's iterative-deepening solution search. Connections between Graphplan's search and IDA* search was first noted by Bonet and Geffner, 1999. One shortcoming of the standard IDA* approach to search is the fact that it regenerates so many of the same nodes in each of its iterations. 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. Given that consecutive iterations of search overlap significantly, we investigated methods for using additional memory to store a trace of the explored search tree in order to avoid repeated re-generation of search nodes. Once we have a representation of the search space that has already been explored, we can transform the way this space is extended in the next iteration. In particular, we can (a) expand the nodes of the current iteration in the order of their heuristic merit (rather than in a default depth first order) and/or (b) we can consider iteratively expanding a select set of states.

Although this type of strategy is too costly to implement in a normal IDA* search, the IDA*-search done by Graphplan is particularly well-suited to these types of changes as the kth level planning graph provides a compact way of representing the search space traversed by the corresponding IDA* search in its kth iteration. Realization of this strategy however does require that we provide an efficient way of extending the search trace represented by the planning graph, starting from any of the search states.

Consider the Figure 1 depiction of the search space for three consecutive Graphplan search episodes leading to a solution for a fictional problem in an unspecified domain. Represented here are just the substates that result from Graphplan's regression search on the ,X,Y,Z, goals, but *not* the mini CSP episodes that attempt to assign actions to each proposition in a state. Thus, each substate on a given planning graph level is linked to it's parent state and is composed of a subset of the parent's goals and the preconditions of the actions that were assigned. In each episode, we show substates generated for the *first* time in a unique shading and use the same shading when the states are regenerated *one planning graph level higher* in the subsequent search episode. A double line box signifies states that eventually end up being part of the plan that is extracted. 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, 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 as defined in (Zimmerman and Kambhampati, 1999).

Certainly Graphplan's search could be made more efficient by using available memory to retain at least some portion of the search experience from episode n to reduce redundant search in episode n+1. This motivation was the focus of the EGBG system (Zimmerman and Kambhampati, 1999), which aggressively recorded the search experience in a given episode in a manner such that essentially *all* redundant effort could be avoided in the next episode. Although that approach was found to run up against memory constraints for larger problems, it suggests a potentially more powerful use for a much more pared-down search trace: leveraging the snapshot view of the entire search space of a Graphplan iteration to focus on the most promising areas. This transformation can free us from the depth-first nature of Graphplan's CSP search, permitting us to move about the search space to visit it's most promising sections first -or even exclusively.

PEGG exploits the search trace it builds, extends, and prunes primarily for its view of the effective search space, and only secondarily to avoid some of the redundant search across episodes. The PEGG algorithm for building and using a search trace retains Graphplan's iterative nature but significantly transforms its search process. We make the following two informal definitions before describing the algorithm developed to transform Graphplan's search:

Search segment: a node-state as generated during Graphplan's regression search from the goal state (which is itself the first search segment), indexed to a specific level of the planning graph. Key content of a search segment S_n at plan graph level k is the proposition list for the state, a pointer to the parent search segment (S_p), and the actions assigned in satisfying the parent segments goals. The last information is needed once a plan is found in order to extract the actions comprising the plan from the search trace.

Search trace (ST): the entire linked set of search segments (states) representing the search space visited in a Graphplan backward search episode. It's convenient to visualize it as 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 ST levels in the reverse order of the plan graph; the top ST 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 search trace will necessarily extend from the highest plan graph level to the initial state, and the plan actions can be extracted from the linked search segments in the ST without unwinding the search calls as Graphplan does.

We also define some processes:

Search trace translation: For a search segment in the ST 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 ST. The fact that search segments are mapped onto the plan graph helps minimize the memory requirements. In order to pickup Graphplan's search from any state in the trace, the number of valid actions for the state goals and their mutex status must be known. The simple expedient of successively linking the search segment to higher plan graph levels in later search episodes makes this bookkeeping feasible.

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 ST) whenever S_p's goals are successfully assigned.
3. Memoize Sp's goals at level j+1 if all attempts to consistently assign them fail.

We claim, without proof here, that as long as *all* the

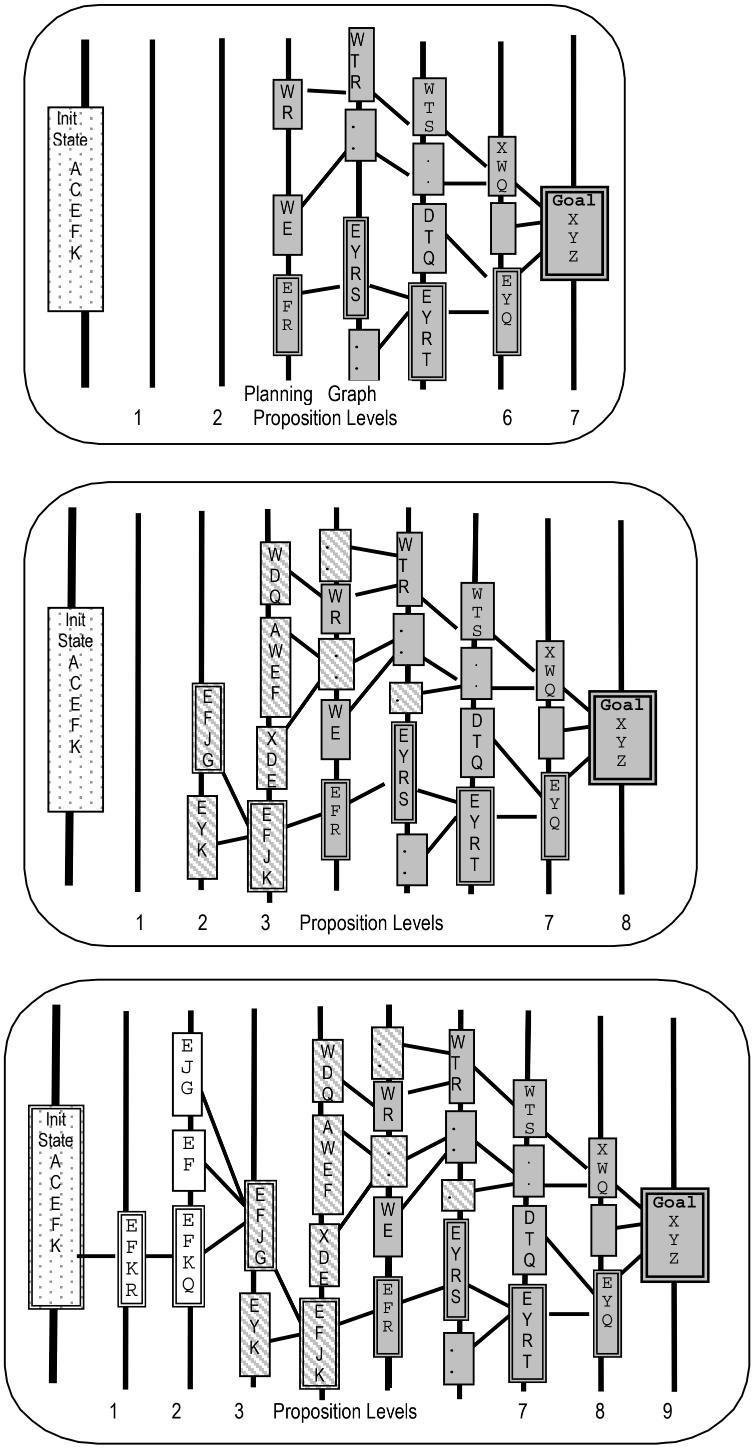

Figure 1. Graphplan's search space: 3 consecutive search episodes on the planning graph

segments in the ST are visited in this manner the planner is guaranteed to find a 'step-optimal' plan in the same search episode as Graphplan (though the number of actions in the plan may differ).

The entire PEGG trace building and search process is detailed [Zimmerman and Kambhampati, 2001, 2002] and we only outline it here. The search process is essentially 2-phased: a promising state from the ST must be selected, then depth-first CSP-type search on the state's subgoals is conducted. If the CSP search fails to find a plan, the planner selects another ST search segment to visit. Our work with a variety of different search trace architectures has highlighted the importance of keeping the search trace small and concise, both due to memory constraints and because the search effort expended in non-solution bearing episodes increases in direct proportion to the number of segments in the ST. We've employ a variety of CSP speedup techniques for the Graphplan style portion of the search process and find that the benefits are compounded because they greatly reduce the number of states visited - and hence tracked in the ST. Chief amongst these methods are explanation based learning (EBL), dependency directed backtracking, domain preprocessing and invariant analysis, and a bi-level plan graph.

As described in [Zimmerman and Kambhampati, 2001, 2002], the search trace provides us with a concise state space view of PEGG's search space, and this allows us to exploit the 'distance based' heuristics employed by state space planners such as HSP-R (Bonet and Geffner, 1999) and AltAlt (Nguyen and Kambhampati, 2000). Two of the approaches for employing these heuristics in PEGG that we have investigated are:

- Ordering the ST search segments according to a given state space heuristic and visiting all of them in order (we term this PEGG-b [2])
- Ordering the ST search segments according to a given state space heuristic and retaining only the 'best' fraction for visitation (PEGG-c)

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 and bounds the size of the search trace that is maintained in memory. As we've reported previously, optimal length plans are generally found, regardless. For this study, Multi-PEGG is run only under the PEGG-b conditions (entire search space visited subject to branch & bound constraints) and we defer further discussion of the PEGG-c to the future work assessment of Section V.

Table 1 compares the performance of PEGG operation to standard Graphplan as well as Graphplan enhanced with the CSP speedup techniques that have been incorporated in PEGG (EBL, DDB, domain preprocessing, etc.). Clearly the enhancements alone have a major impact on standard Graphplan's performance, significantly extending the range of problems it can solve. Focusing on the PEGG-b column its ability to leverage its inter-episodic memory becomes apparent. PEGG-b accelerates planning, by factors of up to 300 over standard Graphplan and 2 - 14x over even the enhanced Graphplan.

When running in this mode, PEGG uses the 'adjusted-sum' distance heuristic described in [Nguyen and Kambhampati, 2000] to move about the search space represented in the ST. Summarizing their description: 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:

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

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. (Due to space considerations, we limit our experimentation here to only this distance heuristic.)

As discussed in [Zimmerman and Kambhampati, 2001, 2002], PEGG-b exhibits speedup over Graphplan in spite of the fact that it revisits (but *doesn't* regenerate) every state that Graphplan generates in each non-solution bearing search episode. One primary sources of its advantage lies in the fact that any state in the ST from the previous episode can be extended in the new episode without incurring the search cost needed to regenerate it. If a state in the deepest levels can be extended to the initial state, we will have found a solution while completely avoiding all the higher level search required to reach it from the top level problem goals. Hereafter we refer to a search trace segment that is visited in the solution episode and extended via backward search to find a valid plan as a *seed segment*. Thus, to the extent that the search heuristic identifies a seed segment deep in the ST in the solution episode, PEGG will greatly shortcut the search in what is often the most costly of Graphplan's iterations.

In the next section, we describe an extension to PEGG that enables the system to find (in most cases) *all* step-optimal plans implicit in a given planning graph. This will prove to be key capability in order for Multi-PEGG to generate plans satisfying multiple optimization criteria.

---

[2] The name scheme for PEGG operating in various modes used in [Zimmerman and Kambhampati, 2001, 2002] is retained here to avoid possible confusion.

Problem	Stnd GP cpu sec	GP-e (enhanced Graphplan) cpu sec (steps/acts)	PEGG-b heuristic: adjsum cpu sec (steps/acts)	PEGG-c 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)	12.2 (18/18)	9.4 (18/18)	87.1 (/ 18)	20.5 (/28)
bw-large-C	~	~	~	60.5 (28/28)	738 (/ 28)	114.9 (/38)
bw-large-D	~	~	~	460.9 (36/36)	2350 (/ 36)	~
Rocket-ext-a	846	39.8 (7/36)	2.8 (7/34)	1.1 (7/34)	43.6 (/ 40)	1.26 (/ 34)
Rocket-ext-b	~	27.6 (7/36)	2.7 (7/34)	2.7 (7/34)	555 (/ 36)	1.65 (/34)
att-log-a	~	31.8 (11/79)	2.6 (11/56)	2.2 (11/62)	36.7 (/56)	2.27(/ 64)
att-log-c	~	~	~	22.9 (12 /57)	53.3 (/ 47)	19.0 (/67)
Gripper-8	~	28.8 (15/23)	16.6 (15/23)	8.0 (15/23)	6.6 (/ 23)	*
Gripper-15	~	~	47.5 (36/45)	16.7 (36/45)	14.1 (/ 45)	16.98 (/45)
Gripper-20	~	~	~	44.8 (40/59)	38.2 (/ 59)	20.92 (/59)
Tower-7	~	114.8 (127/127)	14.3 (127/127)	1.1 (127/127)	7.0 (/127)	*
Tower-9	~	~	118 (511/511)	23.6 (511/511)	121(/511)	*
Mprime-1	17.5	4.8 (4/6)	3.6 (4/6)	2.1 (4/6)	722.6 (/ 4)	79.6 (/ 4)
Mprime-16	~	54.0 (8/13)	35.2 (8/13)	5.9 (4/6)	~	~
8puzzle-1	2444	95.2 (31/31)	39.1 (31/31)	9.2 (31/31)	143.7 (/ 31)	119.5 (/39)
8puzzle-2	1546	87.5 (30/30)	31.3 (30/30)	7.0 (30/30)	348.3 (/ 30)	50.5 (/ 48)
8puzzle-3	50.6	19.7 (20/20)	2.7 (20/20)	1.8 (20/20)	62.6 (/ 20)	63.3 (/ 20)
aips-grid1	312	66.0 (14/14)	34.9 (14/14)	8.4 (14/14)	739.4 (/14)	640.5 (/14)
aips-grid2	~	~	~	129.1 (26/26)	~	~

Table 1 PEGG performance vs. Graphplan, enhanced Graphplan and a BSS heuristic planner

GP-e: Graphplan enhanced with bi-level PG, domain preprocessing, EBL/DDB, goal & action ordering
PEGG-b: Same as PEGG, all segments visited as ordered by adjsum heuristic
PEGG-c: bounded PE search, only best 20% of search segments visited, as ordered by adjsum 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, 256 M RAM
"adjusum2" and "combo" are the most effective heuristics used by AltAlt
~ indicates no solution was found in 30 minutes * indicates problem wasn't run

III Extracting *all* valid plans with PEGG

As discussed in the introduction, extracting all valid plans from even the k-level planning graph, where k is the first level at which a problem solution can be found, is in general intractable for Graphplan. Indeed, no existing planner efficiently does this. We describe here a version of PEGG, which we call PEGG-ap (All Plans) that can in fact efficiently generate all such plans in reasonable time for problems that are not highly solution dense and can stream an arbitrarily large number of them even when there are thousands. It's the combination of PEGG's search trace and the planning graph that make this a feasible proposal for PEGG.

Consider the depiction of Graphplan's search space in the solution episode (third graph) of Figure 1. This corresponds to the ST as it exists immediately after the first plan is found. At this point we've provably shown that each state (set of subgoals) corresponding to the sets of assigned actions in a step of this plan can be extended to the initial state via Graphplan's CSP-style search. These states are the nine speckled search segments in the figure and we will hereafter refer to any such state as a *plan state*. In effect then, such a state at level m can be seen as the root node of a subtree with *at least one* branch that extends from level m to the initial state. We will call such a subtree a *plan stem* (or just stem) and observe that there may be many valid plans implicit in the given planning graph that have the same plan stem as their base. Now consider a planning system which seeks to find all valid plans on a planning graph and that can keep track of such plan stems each time it finds a new plan. If the system can efficiently check during the regression search to see if the set of subgoals, *S*, to be satisfied at a given level m corresponds to one of these states, it has a powerful means of shortcutting that search. Whenever *S* corresponds to one of the plan stem nodes in memory the planner will have found a new plan with a head consisting of the actions/steps assigned in regression search to level m and a tail consisting of the actions/steps corresponding to the plan stem in memory. It can then immediately backtrack in search of other plans.

The fact that PEGG conducts its search on a planning graph suggests an efficient approach for retaining in memory the states associated with a valid plan: the same caches used to memoize states that *cannot* be consistently satisfied during regression search (i.e. 'nogoods') can be used to memoize the states in the extracted plan. Like Graphplan, PEGG's memo-checking routine checks these planning graph level-specific caches anyway before attempting to assign a set of subgoals in CSP fashion. PEGG-ap, has a modified memo saving routine so that when a valid plan is found, it memoizes each plan state at its associated planning graph level, and includes a pointer to the search segment in the ST. The PEGG-ap memo-checking routine differentiates between a nogood memo and a plan state memo such that when a search state matches a plan memo (from some plan already identified), the routine returns a pointer to the relevant search segment in the ST. This enables PEGG-ap to construct a new plan(s) without further search.[3] Note that since all search segments that are part of a valid plan are anyway contained in the ST, it is not necessary to actually store each plan so generated. As long as we maintain a list of the last search segment in a plan tail (i.e. the state whose subgoals are subsumed by the initial state) the upward-linked structure of the ST allows us to extract all identified plans from it on demand.

PROBLEM	TOTAL PLANS	RUN TIME 1ST PLAN	RUN TIME ALL PLANS	SIZE OF ST (no. of states) After 1st plan / After all plans
BW-LARGE-A	1	1.3	2.9	52 / 107
HUGE-FCT	84	9.3	26.6	6642 / 16,728
FERRY6	384	15.8	17.2	377 / 427
GRIPPER8	1680	17.0	32.5	7670 / 10,730
TOWER6	1	1.9	2.3	315 / 440
EIGHT1	12	40.1	75.0	18,650 / 29,909
ROCKET-EXT-A	(>2073)	2.9	(> 14,000)	188 / (238)
ROCKET-EXT-B	1111	1.1	77.0	194 / 2200
ATT-LOG-A	1639	2.9	2407	279 / 818

Table 2 PEGG-ap experiments with extracting *all* plans at the first solution level of the planning graph
Values in parentheses are partial results reported at the time the run was terminated. All planners in Allegro Lisp, runtimes (excl. gc time) in cpu seconds on Pentium III, 900 mhz, Windows 98, 128 M RAM

The performance of PEGG-ap on a sampling of benchmark planning problems is reported in Table 2. The system was set to search in the PEGG-b mode; all ST search segments are ordered and visited according to the 'adjusted-sum' heuristic. The first column of values reports the total number of step-optimal plans generated at the planning graph level at which the first problem solution was found. Clearly, the solution density varies greatly across domains and problems, from the Tower of Hanoi domain that can only have one solution to the logistics domains that may have thousands of valid optimal plans implicit in the planning graph at the solution level. Columns 3 and 4 report run times in cpu seconds to find the first plan and all plans respectively, and the figures testify to the effectiveness of this approach in extracting the remaining plans once the first plan has been found. For example, on the HUGE-FCT problem it takes PEGG-ap 9.3 seconds to generated the first solution and then just over 17 seconds to find the remaining 83 on the 18-level planning graph. The first solution to GRIPPER8 is found in 17 seconds and then the remaining 1679 solutions are generated within another 16 seconds. Many logistics domains problems are so solution dense however that there are thousands of step-optimal plans on the planning graph at the solution level. In the case of ROCKET-EXT-A, for example PEGG-ap had streamed over 2000 plans in 3 ½ hours when the run was terminated.

The fifth column provides a measure of the additional memory required in order for PEGG-ap to extract all step-optimal plans as compared to just the first plan found. We compare here the size of the search trace at the time the first plan is generated with its size after all plans have been found. As expected, the ST grows as more of the states are visited in an attempt to find other plans, but the growth is not linear in the number of plans. This is a reflection of the fact that for most domains/problems plans often share many of the same ST states. The number of search segments (states) in the ST increases by a factor of 11 in the worst case here, but on average the increase is a factor of 2 larger. In no case has this memory demand exceeded the available swap space on the machine used.

IV Streaming plans based on multiple optimization criteria

Up to this point, all versions of PEGG we've discussed are capable of optimizing the number of plan steps. This ability is inherited from the IDA* nature of Graphplan's search process (The connections between Graphplan's search and IDA* was first noted by Bonet and Geffner, 1999.) In order for Multi-PEGG to also handle other optimization criteria, we must have a means of estimating the 'cost' of a achieving a state in terms of the criteria. We start by assigning propositions in the initial states a cost of *zero* and an execution cost for each action. Since PEGG conducts regression search from the problem goals, the cost of reaching those goals from any state generated during the search (e.g. the states in the ST) is easily tracked as the cumulative cost of the assigned actions up to that point. Estimating the cost of reaching a given state from the initial state however, is problematic. To evaluate that cost we need to propagate the costs from the initial state to the state using the mutual dependency between propositions and actions. Specifically, the cost to achieve a proposition

[3] A search segment can be a stem root for more than one valid plan since there may be more than one consistent assignment of actions satisfying its goals.

depends on the cost to execute the actions supporting it, which in turn depends on the costs to achieve propositions that are their preconditions. The planning graph is well suited to represent the relation between propositions and actions, and we will make heavy use of it.

There are two measures of action and state cost that we calculate and propagate in Multi-PEGG:

- *Max cost*: the value of the proposition with the maximum cost in a set of propositions (a state or the preconditions of an action).
- *Sum cost*: the sum of the costs of all propositions in a set

The first measure is most accurate when all preconditions of an action (state) depend on each other and the cost to achieve all of them is equal to the cost to achieve the costliest one. This measure never overestimates the cost and is *admissible*. The second measure is most accurate when a state or all preconditions of an action are independent. Although clearly inadmissible, it has been shown in [11; 2] to be more effective than the *max* measure. Note that the sum cost will always decrease for an action when the cost of one of its preconditions improves, but this is not guaranteed for max cost. As described below we will make use of these measures both separately and in combination in deciding which states to expand during search.

In seeking a 'compromise' estimate of the true cost of reaching the initial state from a given state, we have considered linear combinations of the max and sum measures. Noting that the last two terms of the adjusted-sum heuristic (see section II) provide a measure of the inter-dependence of the propositions in a state, we experimented with using it as a weighting. The following cost estimate for a state *S*, which we will call *adjusted-combo,* has proven effective for the Multi-PEGG search process we will describe below:

$$cst_{adj-combo} = \left[\frac{lev(S) - \max_{p_i \in S} lev(p_i)}{glev(S) - 1}\right] sum(S) + \left[1 - \frac{lev(S) - \max_{p_i \in S} lev(p_i)}{glev(S) - 1}\right] \max(S)$$

where: *lev(S)* and *lev(p)* are as defined in section II, and *glev(S)* is the planning graph level at which state S is currently being evaluated.

Note that since no state *S* will ever be generated in regression search at a planning graph level lower than *lev(S),* the two weighting terms (in brackets) will always sum to one. This cost estimate has the desired property that the higher the degree of negative interactions between the subgoals in *S,* the larger the fraction of the estimate comes from summing the cost of its subgoals. This is clearly an inadmissible heuristic since it can overestimate the cost of a state, but this is of somewhat less concern since Multi-PEGG seeks to stream plans of increasing quality.

We also must confront the issue of normalizing the cost component to the length component when they are combined in a user's linear preference formula. The intent of a preference formula such as α *length* + β *cost* will not be met if there is no base upon which they can be compared. Ideally, we'd like to normalize each component over its optimal value, but in general, we don't know those values. However, as described below, Multi-PEGG in fact first finds a step-optimal plan and then seeks to find a better plan with respect to the user's preference. As such, at the point where it needs a value for plan quality in order to conduct branch and bound search, it has the *optimal* plan length and *one possible* plan cost in hand. When generating the quality value, *q* for a candidate plan we use these base values (*opt-length* and *base-cost* , respectively) to perform a rough normalization of the actual plan parameters (*length* and *cost*) in Multi-PEGG as follows:

$$q = \alpha \frac{\text{length}}{\text{opt–length}} + \beta \frac{\text{cost}}{\text{base–cost}}$$

We can now give an overview of the high-level algorithm used by Multi-PEGG to stream plans that increasingly approach *q,* a specified optimization formula involving more than just plan length:

1. Find the first valid plan -which will be step optimal- using PEGG's approach for conducting search using a search trace. Memoize its constituent states as successful plan states and return the plan to the user. Whenever the planning graph is extended, propagate not only mutex information but also action and proposition *max* and *sum* cost information.
2. With a valid plan in hand, determine it's quality value based on the user-specified criteria, *q*.
3. Define the search space for the next search episode in the following manner: Sort the remaining search segments (states) in the ST based on their *q* criteria. Plan length is set by the current length of the planning graph (say, *k*) and estimates of a state's cost are made based on the propagated cost of its subgoals using the *adjusted-combo* formula.
4. Seek increasingly 'higher quality' plans by conducting branch and bound search (using the *q* value of the best plan found) on the sorted ST states. Any candidate state is *visited* (as defined in section II) as long as its estimated *q* value is less than that of the current best plan. New plans are generated in the manner described for PEGG-ap; either by reaching the initial state or an existing plan state. Whenever the branch and bound finds a lower cost plan, return it to user, memoize its plan states, and update the bounding *q* value.
5. When the branch & bound search space is exhausted at level k, extend the planning graph (propagating cost information), translate the ST up one level, and sort the search states as described in step 3 -with two additions:
   a. Filter from the search space for this episode any state that does not have a decreased *sum* cost

value. (If the cost has not decreased there is no way that it can be extended to a lower cost plan than the current best.)

b. Each state *S,* visited in the previous episode at associated planning graph level *k* that does not extend to a plan effectively provides an updated estimate of *lev(S).* Instead of the original *lev(S)* value, which is the first level at which the propositions are binary non-mutex, we now have an n-ary non-mutex level estimate, which is just *k.*

6. Return to step 4.

This algorithm could of course go on seeking a better plan indefinitely, so in practice we enforce a maximum runtime.

To date Multi-PEGG has been tested on three classical problem domains that we modified to enable testing of its ability to handle multi-criteria.

ROCKET domain

The standard version of this highly parallel logistics domain involves multiple rockets that fly between locales carrying cargo and people. We added cost values to the domain actions as follows:

- rockets' MOVE action> 4
- REFUEL> 3
- LOAD and UNLOAD actions> 1

The benchmark ROCKET-EXT-A and B problems involve 2 rockets, 4 locales, and 10 people and cargo items that must reached goal locations. For both problems Graphplan finds a step-optimal plan of length 7, (which involves using both rockets) but there are a *large* number of such step optimal plans on the 7-level planning graph (see Table 2) and the number of actions in them may vary between 30 and 36. For this fairly simple problem structure it's straightforward to manually determine the optimal plans in terms of actions or cost; if only one rocket is used the goals can be reached in two fewer rocket trips, but it requires one additional plan step. Beyond 8 steps no other cost reductions are achievable.

Table 3 reports on Multi-PEGG's performance in seeking an optimal plan based on different linear combinations of the plan length and plan cost criteria. Here we attempt to give a feel for the dynamic nature of the plan streaming by reporting for each user preference formula , the plan length and cost and its calculated *q* value for the first plan found, and then after 30, 120, and 1200 cpu seconds of runtime. (We don't report values for the *1.0 L + 0 C* formula since this is basic Graphplan's bias. Because cost is absent from the optimization expression, all plans found at the first solution level will have equal 'quality'.) The table reveals several interesting characteristics of Multi-PEGG's search process. Once the first plan is found on the 7-level planning graph, the branch and bound search for a lower cost plan on that graph is quite effective in pruning the search space. Whereas PEGG-ap was still searching for all possible plans at level 7 after 14,000 seconds, Multi-PEGG, after 1200 seconds, completes its search at level 7, extends the planning graph, and conducts search on the 8 level graph for all but the first row optimization criteria. The higher the cost weighting of the criteria, the more the search is pruned on a given planning graph level. The inadmissible nature of the *adjusted-combo* cost heuristic is manifest in the fact that the .8L + .2C and .5L + .5C formulas find some slightly lower cost plans on the 7-level graph than the two formulas with higher cost weightings. However the user's preference appears to be reasonably served for these latter two formulas in that they move on fairly quickly to find some much higher quality (lower *q* value) plans -at least based on *their* criteria- on the 8-level planning graph.

The ROCKET domain problem provides limited exercise for the type of multiple criteria optimizing that Multi-PEGG does, so we look next at a more complex logistics domain involving more than one mode of transportation with different associated costs.

ATT LOGISTICS domain

The standard version of this domain involves two modes of transporting packages; via airplane and via truck. However, the trucks can only operate within a city (hauling packages from the Post Office to the airport) and the airplanes are used to fly between cities. We've extended this domain by not only giving costs to the actions, but enabling trucks to travel between cities that are within range of their fuel capacity. They must refuel at each such city. (For simplicity, we've not introduced actual refueling actions for airplanes, but it's straightforward to do so). The trucks are constrained from traveling directly to any city by

Optimization Criteria L: length C: cost	**1st Plan** [step length/cost]	*q* val	cpu sec.	**Best plan at 30 sec** [step length/cost]	*q* val	**Best plan at 2 min.** [step length/cost]	*q* val	**Best plan at 20 min.** [step length/cost]	*q* val
.8 L + .2 C	[7 / 56]	1.0	3	[7 / 52]	.98	[7 / 52]	.98	[7 / 50]	.97
.5 L + .5 C	[7 / 56]	1.0	3	[7 / 52]	.96	[7 / 52]	.96	[8 / 49]	.95
.2 L + .8 C	[7 / 56]	1.0	3	[7 / 56]	1.0	[7 / 56]	1.0	[8 / 45]	.89
0 L + 1.0 C	[7 / 56]	1.0	3	[7 / 56]	1.0	[8 / 49]	.86	[8 / 45]	.80

Table 3. Multi-PEGG streaming of plans on the ROCKET-EXT-A problem, modified to include action costs.
All planners in Allegro Lisp, runtimes (excl. gc time) in cpu seconds on Pentium III, 900 mhz, Windows 98, 128 M RAM

a 'NEXT-TO' fact added to the 'DRIVE' operator and a set of facts in the initial condition that prescribe which cities are directly next to each other. The cost values for actions are as follows:

- LOAD-TRUCK, UNLOAD-TRUCK> 1
- LOAD-AIRPLANE, UNLOAD-AIRPLANE> 1
- DRIVE-TRUCK1 (local, in-city trip)> 1
- DRIVE-TRUCK2 (inter-city trip)> 3
- REFUEL-TRUCK (needed inter-city only) 1
- FLY> 20

This cost structure is such that, depending on such things as where the truck and package(s) are located in a city, whether their destination is the airport or a post office of a distant city, and how many times a truck must be refueled, transporting the cargo via truck may be cheaper than flying. Note that delivery via truck could also take fewer steps than via airplane because transfer of the cargo from truck to airplane is avoided.

The original benchmark ATT-LOG-A problem that we focus on here features 8 packages to be transported, 3 cities (LA, PGH, BOS) each having one airport and one post office, 1 truck in each city (initially), and both airplanes are in one city. The step-optimal plan for the standard problem is 11 steps and PEGG-ap finds that there are plans ranging from 52 to 76 actions on this 11-level planning graph. (In terms of our introduced cost structure the least cost, 11-step plan would have a value of 128).

Our modified ATT-LOG-A problem retains *all* the original parameters except that we introduce connected cities linking the three destination cities (and thus permitting truck travel) as follows:

- 4 cities between BOS and PGH
- 6 cities between PGH and LA
- 6 cities between BOS and LA

Each of these connecting cities contains an airport (but no post office) so airplanes can also visit them and, feasibly load/unload cargo. We designed the routing structure so that, in combination with the cost structure, truck transportation of cargo will only provided a cost advantage between the cities of BOS and PGH, albeit at the expense of time steps. We note that the additional transportation routes increases the branching factor of this problem considerably, so that although PEGG-ap extracts all step-optimal plans of the original problem within about 40 minutes, it is unable to do so in twice that long on our modified version.

Table 4 reports the performance of Multi-PEGG on this problem for the same optimization formulas and runtime intervals discussed for Table 3. Here there is much greater variation in the quality of the streamed plans due to the more complex structure of the logistical domain. Broadly speaking, the streaming process on this problem has two main phases once the first, step-optimal plan is found; 1) optimizing over the cost of various action sets in alternative 11-step plans 2) searching beyond 11 steps for longer, but less costly plans that use inter-city truck transportation between PGH and BOS instead of airplanes. The branch and bound on plan cost again greatly helps in pruning the search space, as Multi-PEGG begins examining plans of greater than the step-optimal length within 20 minutes for three of the four optimization formulas. For the formulas in the last two rows of the table, Multi-PEGG in fact examines 13-step plans and greatly improves on its least cost 11-step plan by finding some that use the PGH truck to transport three packages to BOS instead of flying them.

The reported results also indicate that, while increasing the bias towards low cost plans causes a more rapid move in this direction for the first two formulas, the trend does not continue with the third formula (compare plan cost trends for these formulas in columns 3 or 4). This is probably due to the complex interactions between how the ST search space is visited (which is directed by the cost heuristic) and the subsequent memoization of both failing nodes and successful plan stem nodes.

V Conclusions and Future Work

We have conducted an investigation into the feasibility of

Optimization Criteria L: length C: cost	**1st Plan** [step length/cost]	*q* val	cpu sec.	**Best plan at 30 sec** [step length/cost]	*q* val	**Best plan at 2 min.** [step length/cost]	*q* val	**Best plan at 20 min.** [step length/cost]	*q* val
.8 L + .2 C	[11 / 208]	1.0	12	[11 / 182]	.98	[11 / 166]	.97	[11 / 128]	.94
.5 L + .5 C	[11/ 208]	1.0	12	[11 / 166]	.95	[11 / 144]	.94	[11 / 128]	.90
.2 L + .8 C	[11 / 208]	1.0	12	[11 / 180]	.96	[11 / 160]	.95	[13 / 111]	.78
0 L + 1.0 C	[11 / 208]	1.0	12	[11 / 166]	.91	[13 / 115]	.80	[13 / 107]	.71

Table 4. Multi-PEGG streaming of plans on the ATT-LOG-A problem, modified to include action costs.

All planners in Allegro Lisp, runtimes (excl. gc time) in cpu seconds on Pentium III, 900 mhz, Windows 98, 128 M RAM streaming parallel plans satisfying multiple criteria using a Graphplan-based planning system. Our preliminary work shows that Multi-PEGG's use of a concise search trace can be exploited to allow it to efficiently generate a stream of plans that monotonically approach a user's preference for plan quality when expressed as a linear preference function on two variables. On the admittedly limited number of problems examined to date, Multi-PEGG is not only capable of finding the least cost step-optimal plan, but it finds longer length plans that come closer to satisfying the multi-objective optimization criteria.

Extending the current system to handle different optimization criteria and more than two does appears to be a straightforward task. Each such criterion requires a suitable estimation function, and the 'cost' values must be propagated in the planning graph separately. However, the approach to ordering states in the ST according to a multi-variable linear preference functions remains unchanged. It is also not a difficult undertaking to extend the type of criteria the user can employ to such things as 'I am not interested in plans costing over x' or 'Give me only plans shorter than length y'.

Overcoming the make-span bias of Multi-PEGG

In spite of the early success of the approach reported in this paper, it clearly has some disadvantages. It inherently starts with a step-optimal plan and, with some help from branch and bound techniques, searches on incrementally longer planning graphs streaming it's current best plan as it does so. If the user's primary plan quality criteria is cost, not length, and the types of low cost plans that are likely to be of interest are many steps longer than the shortest length plan, this approach could be unsatisfactory. Although we recognized this limitation early in the investigation, we also had in mind two major augmentations that might well overcome it, and so proceeded with a test of the simpler system reported here. We discuss these two augmentations to Multi-PEGG next.

Liberation from Graphplan's level-by-level search

There is in fact nothing formidable that *requires* Multi-PEGG to finish its search on a given planning graph level before considering possible plans on extensions of the planning graph. The search trace again proves to be very useful in this regard. Once the first valid plan has been found and a plan quality value established for subsequent branch and bound search, the ST can be translated up any desired number of levels (subject to the ability to extend the graph correspondingly and propagate the cost values) and used in a search for plans of arbitrary length. Referring back to Figure 1, this is equivalent to translating the ST tree of the third search episode pictured upward on the planning graph so that the XYZ root node now lies on some level higher than 9. If we then assess the multi-criteria q values for the search segments (states) in the ST at these higher levels we can co-mingle them with the same search segments from lower levels and *order all of them together* for visitation according to our plan quality formula. To the extent that we have an effective estimation formula for identifying the lowest cost plans, this will essentially enable Multi-PEGG to concurrently consider multiple length plans in its branch and bound search for a better plan.

This would be a prohibitive idea in terms of memory requirements if we had to store multiple versions of the ST, but we can retain only the one version of it and simply store any level-specific cost and heuristic information in its search segments as values indexed to their associated planning graph levels. Interesting problems that arise 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' operators.

Shortcutting the search in a given episode

Of the two modes for employing distance heuristics discussed in section II, we have only reported the performance of Multi-PEGG when it visits *all* states in the ST (i.e. PEGG-b mode), modulo the branch and bound process. It's also possible to augment the branch and bound pruning of search by screening from consideration those states that do not meet some threshold criteria based on a distance heuristic. Such states generated in Graphplan's regression search hold little or no promise of being extended into a solution, yet their inclusion in the search trace means PEGG will have to expand them eventually in each intermediate search episode. We have found that the distance-based heuristics are effective in identifying such states, and have experimented with various threshold options for restricting those maintained in the ST. Although such filtering of the search space forfeits the *guarantee* that PEGG will return a step-optimal solution, in practice we find that that even restricting the active ST to the heuristically best 10-15% of the generated states has no impact on the quality of returned plans. When PEGG operates in this mode, (tagged as 'PEGG-c' in Table 1) there is a dramatic reduction of both the size of the working ST and the time spent in non-solution bearing search episodes. As indicated, PEGG-c solves many more problems than either standard or enhanced Graphplan (GP-e) and exhibits speedups of 40x or more over GP-e where both find solutions. The table also reports the length of the plans produced (in terms of steps and actions). In all cases, PEGG-c finds a plan of equivalent step-length to the Graphplan optimal plan.

The intuition for Multi-PEGG is that, besides looking for the 'next best plan' we only want to visit a search segment in the ST that has a high likelihood of being extended into a valid plan. Of course, this may also screen out some of the longer length but lower cost plans that we may be interested in, so this is an empirical issue that needs to be investigated.

Bounding the length of plans that need to be considered

Another interesting issue associated with this approach to optimizing over multiple criteria is whether we can assess *when* the streaming process can be terminated due to no (or low) possibility of improving on the current best plan. The planning graph may again prove to be a useful structure for deducing such bounds on the search process.

References

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

Zimmerman, T. and Kambhampati, S. 2002. Using memory to transform Graphplan's search. Submitted to Eighteenth National Conference American on Artificial Intelligence, (AAAI 2002).

Zimmerman, T., Kambhampati, S. 2001. Effective Interplay of State Space and CSP Views in a Single Planner. ASU CSE TR-01-008.

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

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