AltAlt: Combining the Advantages of Graphplan and Heuristic State Search

Romeo Sanchez Nigenda, XuanLong Nguyen & Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
Email: {rsanchez,xuanlong,rao}@asu.edu

September 17, 2000

Abstract

Most recent strides in scaling up planning have centered around two competing themes–disjunctive planners, exemplified by Graphplan, and heuristic state search planners, exemplified by HSP and HSP-R. In this paper, we describe a planner called *AltAlt*, which successfully combines the advantages of the two competing paradigms to develop a planner that is significantly more powerful than either of the approaches. *AltAlt* uses Graphplan's planning graph in a novel manner to derive very effective search heuristics which are then used to drive a heuristic state search planner. *AltAlt* is implemented by splicing together implementations of STAN, a state-of-the-art Graphplan implementation, and HSP-r, a heuristic search planner. We present empirical results in a variety of domains that show the significant scale-up power of our combined approach. We will also present a variety of possible optimizations for our approach, and discuss the rich connections between our work and the literature on state-space search heuristics.

1 Introduction

There has been a rapid progress in plan synthesis technology in the past few years, and many approaches have been developed for solving large scale deterministic planning problems. Two of the more prominent approaches are "disjunctive" planners, as exemplified by Graphplan [2] and its many successors including IPP [10] and STAN [13]; and heuristic state search planners exemplified by UNPOP [14], HSP [4] and HSP-R [3].

Graphplan-style systems set up bounded length encodings of planning problems, solve those encodings using some combinatorial workhorse (such as CSP, SAT or ILP solvers), and extend the encoding length iteratively if no solution is found at the current encoding level. State search planners depend on a a variety of heuristics to effectively control a search in the space of world states. These two approaches have generally been seen to be orthogonal and competing. Although both of them have produced quite powerful planning systems, they both do suffer from some important disadvantages. Graphplan-style planners typically need to exhaustively search for plans at every encoding length until a solution is found. This leads to prohibitively large space and time requirements in certain problems. In contrast, state search planners can, in the best case, exhibit a solution with linear space and time. Unfortunately, the existing heuristics for state search planners are unable to handle problems with com-

plex subgoal interactions, making them fail on some domains that Graphplan-style systems are able to handle comfortably.

In this paper, we describe a new hybrid planning system called *AltAlt*[1] that cleverly leverages the complementary strengths of both the Graphplan-style planners and the heuristic state search planners. Specifically, *AltAlt* uses a Graphplan-style planner to generate a polynomial time planning data structure. Using the theory we developed in recent work [9], we extract several highly effective state search heuristics from the planning graph. These heuristics are then used to control a heuristic search planner. *AltAlt* is implemented on top of two highly optimized existing planners–STAN [13] that is a very effective Graphplan style planner is used to generate planning graphs, and HSP-r [3], a heuristic search planner provides an optimized state search engine. Empirical results show that *AltAlt* can be orders of magnitude faster than *both* STAN and HSP-r, validating the utility of hybrid approach.

In the rest of this paper, we discuss the implementation and evaluation of the *AltAlt* planning system. Section 2 starts by providing the high level architecture of the *AltAlt* system. Section 3 briefly reviews the theory behind extraction of state search heuristics [9]. Section 4 discusses a variety of optimizations used in *AltAlt* implementation to drive down the cost of heuristic computation, as well as the state search. Section 5 presents extensive empirical evaluation of *AltAlt* system that demonstrate its domination over both STAN and HSP-r planners. This section also presents experiments to study the cost and effectiveness tradeoffs involved in the computation of *AltAlt*'s planning graph-based heuristics. Section 6 discusses some related work and Section 7 summarizes our contributions.

2 Architecture of *AltAlt*

As mentioned earlier, *AltAlt* system is based on a combination of Graphplan and heuristic state space search technology. The high-level architecture of *AltAlt* is shown in Figure 1. The problem specification and the action template description are first fed to a Graphplan-style planner, which constructs a planning graph for that problem in polynomial time. We use the publicly available STAN implementation [13] for this purpose as it provides a highly memory efficient implementation of planning graph (see below). This planning graph structure is then fed to a heuristic extractor module that is capable of extracting a variety of effective and admissible heuristics, based on the theory that we have developed in our recent work [9]. This heuristic, along with the problem specification, and the set of ground actions in the final action level of the planning graph structure (see below for explanation) are fed to a regression state-search planner. The regression planner code is adapted from HSP-R [3].

To explain the operation of *AltAlt* at a more detailed level, we need to provide some further background on its various components. We shall start with the regression search module. This module starts with the goal state and regresses it over the set of relevant action instances from the domain. An action instance a is considered relevant for a state S if the effects of a give at least one element of S and do not delete *any element* of S. The result of regressing S over a is then $(S \setminus eff(a)) \cup prec(a)$–which is essentially the set of goals that still need to be achieved before the application of a, such that everything in S would have been achieved once a is applied. For each relevant action a, a separate search branch is generated, with result of regressing S over that action as the new state in that branch. Search terminates with success at a node if every literal in the state corresponding to that node is present in the initial state of the problem.

Figure 2 pictorially depicts the initial and final state specification of a simple grid problem, and

[1] A Little of this and a Little of That

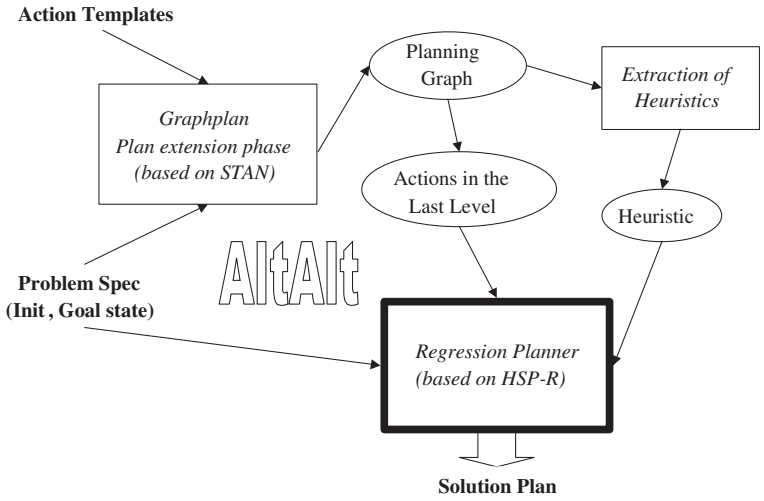
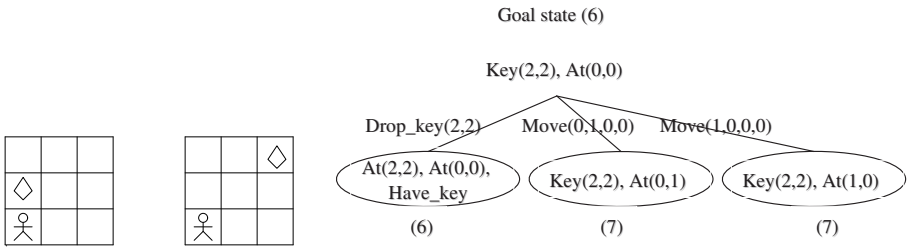


Figure 1: Architecture of *AltAlt*



(a) Grid Problem (b) Regression Search

Figure 2: A simple grid problem and the first level of regression search on it.

presents the first level of the regression search for that problem. In this problem, a robot, in the state (0,0) in the beginning required to pick a key that is in cell (0,1) and place it in the cell (2,2) and get back to its original cell (0,0). The actions in this domain include picking and dropping a key, and moving from one cell to an adjacent cell. If we use the predicate $Key(x, y)$ to represent the location of the key, and $At(x, y)$ to represent the location of the robot, the initial state of the problem is $At(0, 0) \wedge Key(0, 1)$ while the goal state is $Key(2, 2) \wedge At(0, 0)$. Figure 2(b) shows the search branches generated by the regression search. Notice that there is no branch corresponding to a pickup action instance since none of them are relevant for achieving any of the goals in the initial state.

The crux of controlling a regression search involves providing a heuristic function that can estimate the relative goodness of the states on the fringe of the current search tree and guide the search in most promising directions. Such heuristics can be quite tricky to develop. Consider, for example, the fringe states in the search tree of Figure 2(b). Given the robot moves, it is clear that the left most state can never be reached from the initial state–as it requires the robot to be in two positions at the same time. Unfortunately, as we shall show below, naive heuristic functions may actually consider this to be a

more promising state than the other two. In fact, as we discuss in [9], HSP-R, a state-of the art heuristic search regression planner is unable to solve this relatively simple problem!

The issue turns out to be that each one of the three subgoals in the left most state are easier to achieve in isolation than the subgoals of the two other states. Thus any heuristic that considers the cost of achieving subgoals in isolation winds up ranking the left most state as the more promising one. However, once we consider the interactions among the subgoals, the ranking can change quite drastically (as it does in this problem). Taking interactions into account in a principled way turns out to present several technical challenges. Fortunately, our recent work [9] provides an interesting way of leveraging the Graphplan technology to generate very effective heuristics. In the next section, we provide a brief review of this work, and explain how it is used in *AltAlt*.

3 Extraction of Heuristics from Graphplan's Planning Graph

3.1 Structure of the Planning Graph

Graphplan algorithm [2] involves two interleaved stages– expansion of the "planning graph" data structure, and a backward search on the planning graph to see if any subgraph of it corresponds to a valid solution for the given problem. The expansion of the planning graph is a polynomial time operation while the backward search process is an exponential time operation. Since *AltAlt* finds solutions using regression search, our only interest in Graphplan is in its planning graph datastructure.

Figure 3 shows part of the planning graph constructed for the 3x3 grid problem shown in Figure 2. As illustrated here, a planning graph is an ordered graph consisting of two alternating structures, called "proposition lists" and "action lists". We start with the initial state as the zeroth level proposition list. Given a k level planning graph, the extension of the structure to level $k+1$ involves introducing all actions whose preconditions are present in the k^{th} level proposition list. In addition to the actions given in the domain model, we consider a set of dummy "noop" actions, one for each condition in the k^{th} level proposition list (the condition becomes both the single precondition and effect of the noop). Once the actions are introduced, the proposition list at level $k+1$ is constructed as just the union of the effects of all the introduced actions. Planning-graph maintains the dependency links between the actions at level $k+1$ and their preconditions in level k proposition list and their effects in level $k+1$ proposition list.

The critical asset of the planning graph, for our purposes, is the efficient marking and propagation of mutex constraints during the expansion phase. The propagation starts at level 1, with the actions that are statically interfering with each other (i.e., their preconditions and effects are inconsistent) labeled mutex. Mutexes are then propagated from this level forward by using two simple propagation rules: Two propositions at level k are marked mutex if all actions at level k that support one proposition are pair-wise mutex with all actions that support the second proposition. Two actions at level $k+1$ are mutex if they are statically interfering or if one of the propositions (preconditions) supporting the first action is mutually exclusive with one of the propositions supporting the second action. Figure 3 shows a part of the planning graph for the robot problem specified in Figure 2. The curved lines with x-marks denote the mutex relations.

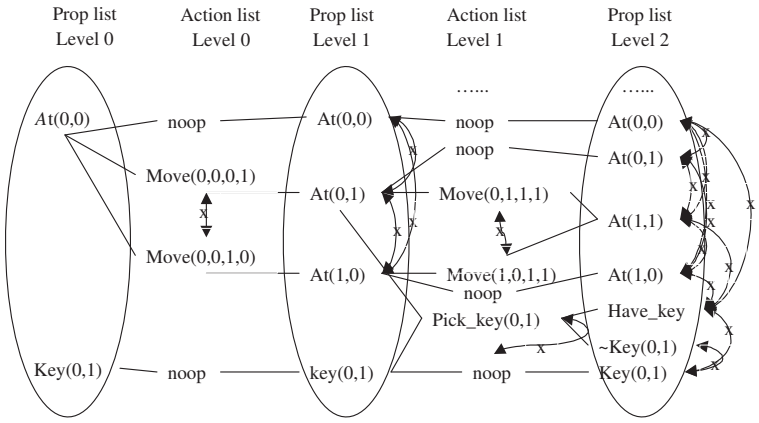


Figure 3: Planning Graph for the 3x3 grid problem

3.2 Heuristics based on the planning graph

To guide a regression search in the state space, a heuristic function needs to evaluate the cost of some set S of subgoals, comprising a regression state, from the initial state–in terms of number of actions needed to achieve them from the initial state. We now discuss how such a heuristic can be computed from the planning graph.

Normally, the planning graph datastructure supports "parallel" plans–i.e., plans where at each step more than one action may be executed simultaneously. Since we want the planning graph to provide heuristics to the regression search module, which generates sequential solutions, we first make a modification to the algorithm so that it generates "serial planning graph." A *serial planning graph* is a planning graph in which, in addition to the normal mutex relations, every pair of non-noop actions at the same le ˙ id ˙ fl –dad «Ï,⁄„ Ł¸á

and expand a serial planning graph until it levels off. (As we shall see later, we can relax the requirement of growing the planning graph to level-off, if we can tolerate a graded loss of informedness of heuristics derived from the planning graph.) We will start with the notion of level of a set of propositions:

Definition 1 (Level) *Given a set S of propositions, denote $lev(S)$ as the index of the first level in the leveled serial planning graph in which all propositions in S appear and are non-mutexed with one another. If S is singleton, then $lev(S)$ is just the index of the first level where the singleton element occurs. If no such level exists, then $lev(S) = \infty$ if the planning graph has been grown to level-off, and $lev(S) = l + 1$, where l is the index of the last level that the planning graph has been grown to (i.e not until level-off). Similarly, denote $lev(p)$ as the index of the first level that a proposition p comes into the planning graph.*

The intuition behind this definition is that the level of a literal p in the planning graph provides a lower bound on the number of actions required to achieve p from the initial state. Using this insight, a simple way of estimating the cost of a set of subgoals will be to sum their levels.

Heuristic 1 (Sum heuristic) $h(S) := \sum_{p \in S} lev(\{p\})$

The sum heuristic is very similar to the greedy regression heuristic used in UNPOP [14] and the heuristic used in the HSP planner [4]. Its main limitation is that the heuristic makes the implicit assumption that all the subgoals (elements of S) are independent. For example, the h_{sum} heuristic winds up ranking the left most state of the search tree in Figure 2(b) as the most promising among the three fringe states. Sum heuristic is neither admissible nor particularly informed. Specifically, since subgoals can be interacting negatively (in that achieving one winds up undoing progress made on achieving the others), the true cost of achieving a pair of subgoals may be more than the sum of the costs of achieving them individually. This makes the heuristic inadmissible. Similarly, since subgoals can be positively interacting in that achieving one winds up making indirect progress towards the achievement of the other, the true cost of achieving a set of subgoals may be lower than the sum of their individual costs. To develop more effective heuristics, we need to consider both positive and negative interactions among subgoals in a limited fashion.

In [9], we discuss a variety of ways of capturing the negative and positive interactions into the heuristic estimate using the planning graph structure, and discuss their relative tradeoffs. One of the best heuristics according to that analysis was a heuristic called $h_{AdjSum2M}$. We adopted this heuristic as the default heuristic in *AltAlt*. In the following, we briefly describe this heuristic.

The basic idea of $h_{AdjSum2M}$ is to adjust the sum heuristic to take positive and negative interactions into account. This heuristic approximates the cost of achieving the subgoals in some set S as the sum of the cost of achieving S, while considering positive interactions and ignoring negative interactions, plus the penalty for ignoring the negative interactions. The first component can be computed as the length of a "relaxed plan" for supporting S, which is extracted by *ignoring all the mutex relations*. To approximate the penalty induced by the negative interactions alone, we proceed with the following argument. Consider any pair of subgoals $p, q \in S$. If there are no negative interactions between p and q, then $lev(\{p, q\})$, the level at which p and q are present together is exactly the maximum of $lev(p)$ and $lev(q)$. The degree of negative interaction between p and q can thus be quantified by:

$$\delta(p, q) = lev(\{p, q\}) - max\,(lev(p), lev(q))$$

We now want to use the δ-values to characterize the amount of negative interactions present among the subgoals of a given set S. If all subgoals in S are pair-wise independent, clearly, all δ values will be zero, otherwise each pair of subgoals in S will have a different value. The largest such δ value among any pair of subgoals in S is used as a measure of the negative interactions present in S in the heuristic $h_{AdjSum2M}$. In summary, we have

Heuristic 2 (Adjusted heuristic 2M) $h_{AdjSum2M}(S) := length(Relaxedplan(S)) + max_{p,q \in S}\ \ \delta(p, q)$

The analysis in [9] shows that this is one of the more robust heuristics in terms of both solution time and quality. This is thus the default heuristic used in *AltAlt*.

4 Implementational issues of extracting heuristics from planning graph

While we described the main components and design issues underlying *AltAlt* system, there are several optimization issues that still deserve attention. Two of them are discussed in this section.

4.1 Controlling the Cost of Computing the Heuristic

The first issue is the cost of computing the heuristic using planning graphs. Although, as we mentioned earlier, planning graph construction is a polynomial time operation, it does lead to relatively high time and space consumption in many problems. The main issues are the sheer size of the planning graph, and the cost of marking and managing mutex relations. Fortunately, however, there are several possible ways of keeping the heuristic computation cost in check. To begin with, one main reason for basing *AltAlt* on STAN rather than other Graphplan implementations is that STAN provides a particularly compact and efficient planning graph construction. In particular, as described in [13], STAN uses a very compact bi-level representation planning graph which exploits the redundancy in the planning graph. Secondly, STAN uses efficient datastructures to mark and manage the "mutex" relations.

While the use of STAN system reduces planning graph construction costs significantly, heuristic computation cost can still be a large fraction of the total run time. For example, in one of the benchmark problems, `bw-large-d`, the heuristic computation takes 3.5 m.sec. while the search takes 3 m.sec. Worse yet, in some domains such as the Sched World from the AIPS 2000 competition suite [1], the graph construction phase winds up overwhelming the memory of the system.

Thankfully, however, by trading off heuristic quality for reduced cost, we can aggressively limit the heuristic computation costs. Specifically, in the previous section, we discussed the extraction of heuristics from a completed leveled planning graph. Since *AltAlt* does not do any search on the planning graph directly, there is no strict need to use the full leveled graph to preserve completeness. Informally, *any subgraph* of the full leveled planning graph can be gainfully utilized as the basis for the heuristic computation. There are at least three ways of computing a smaller subset of the leveled planning graph:

1. Grow the planning graph to some length that is less than the length where it levels off. For example, we may grow the graph until the top level goals of the problem are present without any mutex relations in the final proposition level of the planning graph.

2. Spend only limited time on marking mutexes on the planning graph.

3. Introduce only a subset of the "applicable" actions at each level of the planning graph. For example, we can exploit the techniques such as RIFO [15] and identify a subset of the action instances in the domain that are likely to be "relevant" for solving the problem.

Any combination of the above three techniques can be used to limit the space and time resources expended on computing the planning graph. What is more, it can be shown that the admissibility and completeness characteristics of the heuristic will remain unaffected as long as we do not use the third approach (recall that the definition of level in section 3.2 avoids assigning ∞ as the cost of a set if the underlying planning graph is not grown to level off). Only the informedness of the heuristic is affected. We shall see in the next section that in many problems the loss of informedness is more than offset by the improved time and space costs of the heuristic.

4.2 Limiting the Branching Factor of Regression Search Using Planning Graphs

Although the preceding discussion focused on the use of the planning graphs for computing the heuristic in *AltAlt*, from Figure 1, we see that planning graph is also used to pick the action instances considered in expanding the regression search tree. The advantages of using the action instances from the planning graph are that in many domains there are a prohibitively large number of ground action instances, only a very small subset of which are actually applicable in any state reachable from the initial

state. Using all such actions in regression search can significantly increase the cost of node expansion (and may, on occasion, lead the search down the wrong paths). In contrast, the action instances present in the planning graph are more likely to be applicable in states reachable from the initial state.

The simplest way of picking action instances from the planning graph is to consider all action instances that are present in the final level of the planning graph. If the graph has been grown to level off, it can be proved that limiting regression search to this subset of actions is guaranteed to preserve completeness. A more aggressive selective expansion approach, that we call `sel-exp`, involves the following. Suppose l is the first level at which the literals of the top-level goal are all present in the planning graph without being pairwise mutex. `Set-exp` approach involves expanding the planning graph up to level l, as opposed to level-off. Suppose that we are trying to expand a state S in the regression search, then only the set of actions A_{l-1} is considered to regress the state S. The intuition behind `sel-exp` strategy is that the actions in A_{l-1} comprise the actions that are likely to achieve the subgoals of S in the most direct way from the initial state. As we shall see in the next section, the `sel-exp` strategy, while theoretically incomplete, can have a significant effect on the performance of *AltAlt* in some domains such as the Schedule World[1].

5 Evaluating the Performance of *AltAlt*

AltAlt system as described in the previous sections has been fully implemented. Its performance on many benchmark problems, as well as the test suite used in the recent AIPS-2000 planning competition, is remarkably robust. Our initial experiments suggest that *AltAlt* system is competitive with some of the best systems that participated in the AIPS competition [1]. The evaluation studies presented in this paper are however aimed at establishing two main facts: First, *AltAlt* convincingly outperforms both Graphplan (STAN) and HSP-r systems that it is based on in a variety of domains. Second, *AltAlt* is able to reduce the cost of its heuristic computation with very little negative impact on the quality of the solutions produced.

Our experiments were all done on a Linux system running on a 500 megahertz pentium III CPU with 256 megabytes of RAM. We compared *AltAlt* with the latest versions of both STAN and HSP-r system running on the same hardware. HSP2.0 is a recent variant of the HSP-r system that opportunistically shifts between regression search (HSP-r) and progression search (HSP). We also compare *AltAlt* to HSP2.0. The problems used in our experiments come from a variety of domains, and were derived primarily from the AIPS-2000 competition suites [1], but also contain some other benchmark problems known in the literature. Unless noted otherwise, in all the experiments, *AltAlt* was run with the heuristic $h_{AdjSum2M}$, and with a planning graph grown only until the first level where top level goals are present without being mutex (see discussion in Section 4.1). Only the action instances present in the final level of the planning graph are used to expand nodes in the regression search (see Section 4.2).

Table 1 shows some statistics gathered from head-on comparisons between *AltAlt*, STAN, HSP-r and HSP2.0 across a variety of domains. For each system, the table gives the time taken to produce the solution, and the length (measured in the number of actions) of the solution produced. Dashes show problem instances that could not be solved by the corresponding system under a time limit of 10 minutes. We note that *AltAlt* demonstrates robust performance across all the domains. It *decisively outperforms* STAN and HSP-r in most of the problems, easily solving both those problems that are hard for STAN as well as those that are hard for HSP-r. We also note that the quality of he solutions produced by *AltAlt* is as good or better than those produced by the other two systems in most problems. The table also shows a comparison with HSP2.0. While HSP2.0 predictably outperforms HSP-r, it is

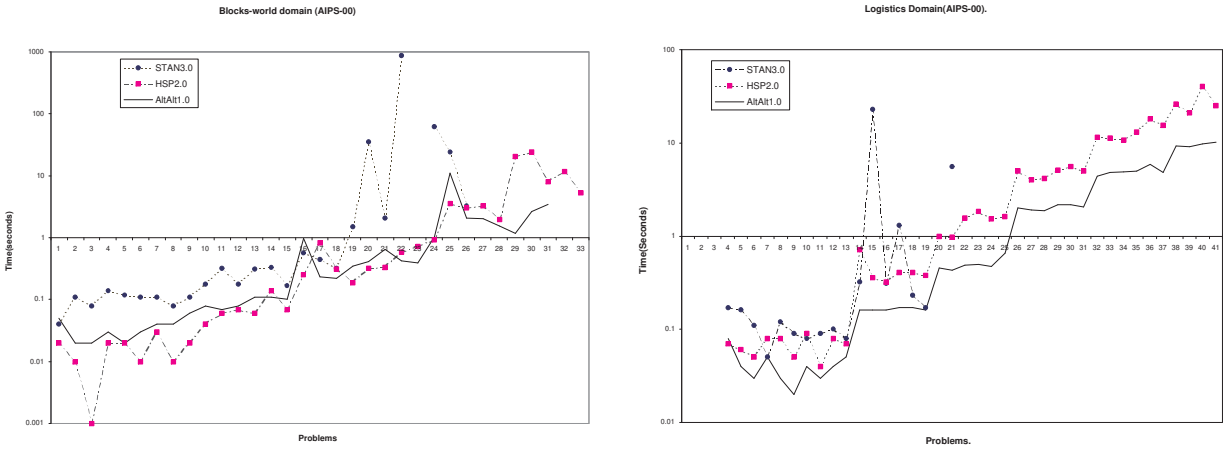
	STAN3.0		HSP-r		HSP2.0		AltAlt(AdjSum2M)	
Problem	Time	Length	Time	Length	Time	Length	Time	Length
gripper-15	-	-	0.12	45	0.19	57	0.31	45
gripper-20	-	-	0.35	57	0.43	73	0.84	57
gripper-25	-	-	0.60	67	0.79	83	1.57	67
gripper-30	-	-	1.07	77	1.25	93	2.83	77
tower-3	0.04	7	0.01	7	0.01	7	0.04	7
tower-5	0.21	31	5.5	31	0.04	31	0.16	31
tower-7	2.63	127	-	-	0.61	127	1.37	127
tower-9	108.85	511	-	-	14.86	511	48.45	511
8-puzzle1	37.40	31	34.47	45	0.64	59	0.69	31
8-puzzle2	35.92	30	6.07	52	0.55	48	0.74	30
8-puzzle3	0.63	20	164.27	24	0.34	34	0.19	20
8-puzzle4	4.88	25	1.35	26	0.46	42	0.41	24
aips-grid1	1.07	14	-	-	2.19	14	0.88	14
aips-grid2	-	-	-	-	14.06	26	95.98	34
mystery2	0.20	9	84.00	8	10.12	9	3.53	9
mystery3	0.13	4	4.74	4	2.49	4	0.26	4
mystery6	4.99	16	-	-	148.94	16	62.25	16
mystery9	0.12	8	4.8	8	3.57	8	0.49	8
mprime2	0.567	13	23.32	9	20.90	9	5.79	11
mprime3	1.02	6	8.31	4	5.17	4	1.67	4
mprime4	0.83	11	33.12	8	0.92	10	1.29	11
mprime7	0.418	6	-	-	-	-	1.32	6
mprime16	5.56	13	-	-	46.58	6	4.74	9
mprime27	1.90	9	-	-	45.71	7	2.67	9

Table 1: Comparing the performance of *AltAlt* with STAN, a state-of-the-art Graphplan system, and HSP-R, a state-of-the-art heuristic state search planner.

still dominated by *AltAlt*, especially in terms of solution quality.

The plots in Figure 4 and Figure 5 compare the time performance of STAN, *AltAlt* and HSP2.0 in specific domains. Plot a summarizes the problems from blocks world and the plot b refers to the problems from logistics domain. The plot in Figure 5 refers to the problems from the scheduling world. These are three of the standard benchmark domains that have been used in the recent planning competition [1]. The y-axis in all these plots is in logarithmic scale. We see that in all domains, *AltAlt* clearly dominates STAN. It dominates HSP2.0 in logistics and is very competitive with it in blocks world. Scheduling world was a very hard domain for most planners in the recent planning competition [1]. We see that *AltAlt* scales much better than both STAN and HSP2.0. (Recall, once again, that HSP2.0 uses a combination of progression and regression search. Comparison with HSP-r system would be even more decisively in favor of *AltAlt*.) Although not shown in the plots, the length of the solutions found by *AltAlt* in all these domains was as good or better than the other two systems.

Evaluating Cost/quality Tradeoffs in the heuristic computation: We mentioned earlier that in all these experiments we used a partial (non-leveled) planning graph that was grown only until all the goals are present and are non-mutex in the final level. As the discussion in Section 4.1 showed, deriving heuristics from such partial planning graphs trades cost of the heuristic computation with quality. To get an an idea of how much of a hit on solution quality we are taking, we ran experiments comparing the same heuristic $h_{AdjSum2M}$ derived once from full leveled planning graph, and once from the partial



(a) Blocks World

(b) Logistics

Figure 4: Results in Blocks world and Logistics

planning graph stopped at the level where goals first become non-mutexed.

The plots in Figure 6 show the results of experiments with a large set of problems from the scheduling domain. Plot a shows the total time taken for heuristic computation and search together, and plot b shows the length of the solution found for both strategies. We can see very clearly that if we insist on full leveled planning graph, we are unable to solve problems beyond 81, while the heuristic derived from the partial planning graph scales all the way to 111 problems. The time taken by the partial planning graph strategy is significantly lower, as expected. Plot b shows that even on the problems that are solved by both strategies, we do not incur any appreciable loss of solution quality because of the use of partial planning graph. This validates our contention in Section 4.1 that the heuristic computation cost can be kept within limits. It should be mentioned here that the planning graph computation cost depends a lot upon domains. In domains such as Towers of hanoi, where there are very few irrelevant actions, the full and partial planning graph strategies are almost indistinguishable in terms of cost. In contrast, domains such as grid world and scheduling world incur significantly higher planning graph construction costs, and thus benefit more readily by the use of partial planning graphs.

6 Related work

As we had already discussed in the paper, by its very nature, *AltAlt* has obvious rich connections to the existing work on Graphplan [2, 13, 10] and heuristic state search planners [4, 3, 14, 16]. The idea of using the planning graph to select action instances to focus the regression search is similar to techniques such as RIFO [15], that use relevance analysis to focus progression search. As discussed in [9], there are several rich connections between our strategies for deriving the heuristics from the planning graphs, and recent advances in heuristic search, such as pattern databases [5], and capturing subproblem interactions [11, 12]. Finally, given that the informedness of our heuristics is closely related to the subgoal interaction analysis, pre-processing and consistency enforcement techniques,

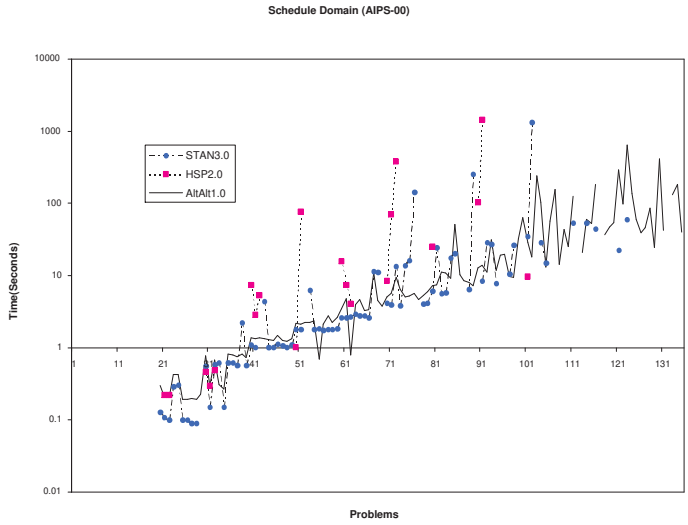
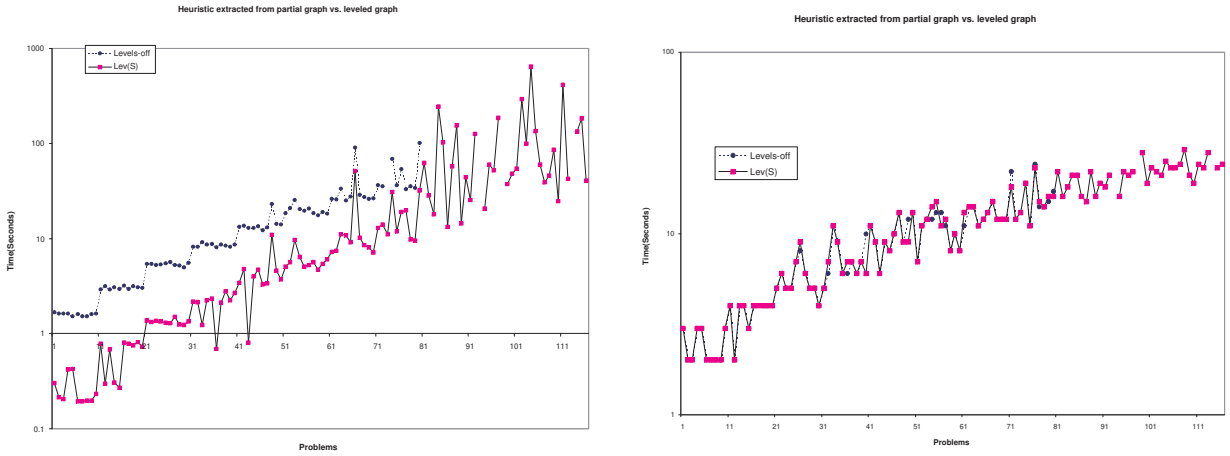


Figure 5: Results in the Scheduling World



(a) Running Time

(b) Solution Quality

Figure 6: Results on trading heuristic quality for cost by extracting heuristics from partial planning graphs.

such as those described in [7, 17, 6] can be used to further improve the informedness of the heuristics.

7 Concluding Remarks

We described the implementation and evaluation of a novel plan synthesis system, called *AltAlt*. *AltAlt* is designed to exploit the complementary strengths of two of the currently popular competing approaches for plan generation–Graphplan, and heuristic state search. It uses the planning graph to derive effective heuristics that are then used to guide heuristic state search. The heuristics derived from the planning graph do a better job of taking the subgoal interactions into account and as such are significantly more effective than existing heuristics. *AltAlt* was implemented on top of two state of the art planning systems–STAN3.0 a Graphlan-style planner, and HSP-r, a heuristic search planner. Our extensive empirical evaluation shows that *AltAlt* convincingly outperforms both STAN3.0 and HSP-r. In fact, *AltAlt*'s performance is very competitive with the planning systems that took part in the recent AI Planning Competition [1]. Our empirical results also show that there exist attractive approaches for trading cost for quality in the computation of planning graph based heuristic.

References

[1] F. Bacchus. Results of the AIPS 2000 Planning Competition. URL: http://www.cs.toronto.edu/aips-2000.

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

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

[4] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proc. AAAI-97*, 1997.

[5] J. Culberson and J. Schaeffer. Pattern Databases. *Computational Intelligence*, Vol. 14, No. 4, 1998.

[6] M. Fox and D. Long. Automatic inference of state invariants in TIM. *JAIR*. Vol. 9. 1998.

[7] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proc. AAAI-98*, 1998.

[8] J. Hoffman. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. Technical Report No. 133, Albert Ludwigs University.

[9] A paper by the authors published in the Proceedings of AAAI-2000. Details suppressed to facilitate blind review.

[10] J. Kohler, B. Nebel, and Y. Dimopoulos. Extending planning graphs to an adl subset. In *Proc. ECP-97*, 1997.

[11] R. Korf and L. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proc. AAAI-96*, 1996.

[12] R. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proc. AAAI-97*, 1997.

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

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

[15] B. Nebel, Y. Dimopoulos and J. Koehler. Ignoring irrelevant facts and operators in plan generation. *Proc. ECP-97*.

[16] I. Refanidis and I. Vlahavas. GRT: A domain independent heuristic for strips worlds based on greedy regression tables. In *Proc. ECP-99*, 1999.

[17] J. Rintanen. An iterative algorithm for synthesizing invariants. In *Proc. AAAI-2000*, 2000.