Planning Graph as the Basis for Deriving Heuristics for Plan Synthesis by State Space and CSP Search

XuanLong Nguyen, Subbarao Kambhampati & Romeo S. Nigenda
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
Email: {xuanlong,rao,rsanchez}@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 UNPOP, HSP and HSP-R. In this paper, we present a novel approach for successfully harnessing the advantages of the two competing paradigms to develop planners that are significantly more powerful than either of the approaches. Specifically, we show that the planning graph structure that Graphplan builds in polynomial time, provides a rich substrate for deriving a family of highly effective heuristics for guiding state space search as well as CSP-style search. The main leverage provided by the planning graph structure is a systematic and graded way to take subgoal interactions into account in designing state space heuristics. For state space search, we develop several families of heuristics, some aimed at search speed and others at optimality of solutions, and analyze many approaches for improving the cost-quality tradeoffs offered by these heuristics. Our normalized empirical studies show that our heuristics handily out-perform the existing state space heuristics. For CSP-style search, we describe a novel way of using the planning-graph structure to derive highly effective variable and value ordering heuristics. We show that these heuristics can be used to improve Graphplan's own backward search significantly. To demonstrate the effectiveness of our approach *vis a vis* the state-of-the-art in plan synthesis, we present and evaluate *AltAlt*, an implementation of our approach, on the suite of problems used in the recent AIPS-2000 planning competition. The results place *AltAlt* in the top tier of the competition planners–outperforming both Graphplan-based and heuristic search based planners.

1 Introduction

The last few years have seen a number of attractive and scalable approaches for solving deterministic planning problems. Prominent among these are "disjunctive" planners, exemplified the Graphplan algorithm of Blum & Furst [3], and heuristic state space planners, exemplified by McDermott's UNPOP [34] and Bonet & Geffner's HSP-R planners [4, 5]. The Graphplan algorithm can be seen as solving the planning problem using CSP techniques. A compact CSP encoding of the planning problem is generated using a polynomial-time data structure called "planning graph" [17]. On the other hand, UNPOP, HSP, HSP-R are simple state space planners where the world state is considered explicitly. These planners rely on heuristic estimators to evaluate the goodness of children states. As such, it is not surprising that heuristic state search planners and Graphplan-based planners are generally seen as orthogonal approaches [44].

Indeed, the sources of strength (as well as weaknesses) of Graphplan and heuristic state search planners are quite different. By posing the planning problem as a CSP and exploiting the internal structure of the state, Graphplan is good in dealing with problems where there are a lot of interactions between actions and subgoals. Also, Graphplan guarantees, theoretically, to find a solution if such a solution exists. On the other hand, having to exhaust the whole search space up to the solution bearing level is a big source of inefficiency of the Graphplan.

In UNPOP, HSP and HSP-R, the heuristic can be seen as estimating the number of actions required to reach a state (either from the goal state or the initial state). To make the computation tractable, these heuristic estimators make strong assumptions about the independence of subgoals. Because of these assumptions, state search planners often thrash badly in problems where there are strong interactions between subgoals. Furthermore, these independence assumptions also make the heuristics inadmissible, precluding any guarantees about the optimality of solutions found. In fact, the authors of UNPOP and HSP/HSP-R planners acknowledge that taking the subgoal interactions into account in a tractable fashion to compute more robust and/or admissible heuristics remains a challenging problem [34, 5].

In this paper, we provide a way of successfully combining the advantages of the Graphplan and heuristic state search approaches. The main idea is to use the planning graph data structure as the basis for deriving effective heuristics for both state search and CSP style search. We list our specific contributions below.

1. We will show that the planning graph is a rich source for deriving effective and admissible heuristics for controlling state space search. Specifically, we describe how a large family of heuristics–with or without admissibility property–can be derived in a systematic manner from a *leveled off* planning graph. These heuristics are then used to control a regression search in the state space to generate plans. The effectiveness of these heuristics is directly related to the fact that they give a better account of the subgoal interactions. We show how the propagated of binary (and higher order) mutual exclusion relations make the heuristics derived from the planning graph more sensitive to negative subgoal interactions. Positive interactions are accounted for by exploiting structure of the planning graph that explicitly shows when achieving a subgoal also, as a side effect, achieves another subgoal. We provide a thorough cost-benefit evaluation of these heuristics:

   - We provide a comparative analysis of the tradeoffs offered by the various heuristics derived from the planning graph. We also show that our heuristics are superior to known hsp-style heuristics (e.g. HSP-r [4]) for planning problems.
   - We consider and evaluate the costs and benefits involved in using planning graphs as the basis for deriving state space heuristics. We point out that planning graphs, in addition to supporting heuristic generation, also help in significantly reducing the branching factor of the state space search by helping it focus on the applicable actions. As for the heuristic generation cost, in addition to adapting several existing optimizations for generating planning graphs efficiently, we also demonstrate that it is possible to cut cost while keeping the effectiveness of the heuristics largely intact by working with partially grown planning graphs.

2. To show that our approach for deriving heuristics from planning graphs is competitive with the current state-of-the-art plan synthesis systems, we implement our ideas in a planner called *AltAlt* and compare *AltAlt*'s performance on the benchmark problems used in the recent AIPS-2000 Planning Competition [1]. Our results indicate that *AltAlt*'s performance on these problems puts it squarely in the top tier of two or three best planners in the competition.

3. To further clarify the connections between heuristic state-search and Graphplan, we show that the distance based heuristics, such as those we derive from the planning graph, can also be used to control Graphplan's own backward search. It is well known that Graphplan's backward search can be seen as a variation of standard systematic backtracking search for constraint satisfaction problems [16]. In order to improve this search, we show how the planning graph can be used to derive more effective variable

and value ordering heuristics for backward search. Of specific interest is our empirical observation that armed with these heuristics, Graphplan can largely avoid its unproductive exhaustive searches in the non-solution bearing levels by starting with planning graphs that are longer than the minimal length solutions. Surprisingly, our heuristics make the search on such longer planning graphs both very efficient, and near-optimal (in that the quality of plans produced are close to the ones produced by the original Graphplan).

4. Our work also makes several pedagogical contributions. To begin with, we point out that the ideas inherent in Graphplan style systems, and the heuristic state search planners are *complementary* rather than competing. We also point out that planning graph based heuristics developed in our work are closely related to "memory based heuristics" (such as pattern databases) [30] that are currently popular in search community. Given that the planning graph can be seen as a representation of the CSP encoding of the planning problem, and mutex propagation can be seen as a form of consistency enforcement, our work also foregrounds the interesting connections between the degree of (local) consistency of the underlying CSP encoding of the planning problem, and the effectiveness of the heuristics generated from the encoding.

The rest of the paper is organized as follows. Section 2 reviews the state space search approach in planning, exemplified by a state search planner such as HSP-R. We discuss the limitations of the heuristic estimators in planners of this approach. Section 3 discusses how the Graphplan's planning graph can be used to measure the subgoal interactions. Section 4 develops several families of effective heuristics, which aims at search speed without insisting on the admissibility. Section 5 focuses on generating the admissible heuristic for optimal planning. Several implementational issues of computing the heuristic functions is investigated in section 6. Section 7 describes *AltAlt*, a planning system based on our ideas, and presents a comparative evaluation of its performance *vis a vis* other state of the art planners. In section 8 we show how the planning graph can be used to develop highly effective variable and value ordering heuristics for CSP encodings for planning problems. We demonstrate their effectiveness by using them to improve the efficiency of Graphplan's own backward search. Section 9 discusses the related work, and section 10 summarizes the contributions of our work.

2 Planning as state space search and the heuristics

Planning can be naturally seen as a search through the space of world states[36]. In this paper, we consider the simple STRIPS representation of classical planning, where each state is represented as a set of propositions (or subgoals). We are given a complete initial state S_0, goal G, which is a set of subgoals and can be incomplete, a set of deterministic actions Ω. Each action $a \in \Omega$ has a precondition list, add list and delete list, denoted as $Prec(a), Add(a), Del(a)$, respectively. The planning problem is concerned with finding a plan, e.g a sequence of action in Ω, that when applied to the initial state S_0 (and executed) will achieve the goal G.

In progression state space search, an action a is said to be applicable to state S if $Prec(a) \subseteq S$. The result of the progression of S over an applicable action a is defined as:

$$Progress(S, a) = S + Add(a) - Del(a)$$

The heuristic function over a state S is the cost estimate of a plan that achieves G from the state S.

In regression state space search, an action a is said to be applicable to state S if $Add(a) \cap S \neq \phi$ and $Del(a) \cap S = \phi$. The regression of S over an applicable action a is defined as:

$$Regress(S, a) = S + Prec(a) - Add(a)$$

The heuristic function over a state S is the cost estimate of a plan that achieves S from initial state S_0.

The efficiency of state search planners and the quality of the solution that they return depend critically on the informedness and admissibility of these heuristic estimators, respectively. The difficulty of achieving the desirable informedness and admissibility of the heuristic estimates is due to the fact that subgoal interact in complex ways. There are two kinds of subgoal interactions: Negative interactions and positive interactions. Negative interactions happen when achieving one subgoal interferes with the achievement of some other subgoal. Ignoring this kind of interactions would normally underestimate the cost, rendering the heuristic uninformed. Positive interactions happen when achieving one subgoal also makes it easier to achieve other subgoals. Ignoring this kind of interactions would normally overestimate the cost, rendering the heuristic estimate inadmissible.

For the rest of this section we will demonstrate the importance of accounting for subgoal interactions in order to compute informed and/or admissible heuristic functions. We do so by specifically examining the weakness of heuristics that ignore these subgoal interactions, as shown by a typical state search planner such as HSP-R [4]. Similar arguments can also be generalized over other state space planners such as HSP, UNPOP, etc.

Recall that HSP-R cast the planning problem as search in the *regression* space of the world states. The heuristic value of a state S is the estimated cost (number of actions) needed to achieve S from the initial state. It is important to note that since the cost of a state S is computed from the initial state and we are searching backward from the goal state, the heuristic computation is done only once for each state. Then, HSP-R follows a variation of A* search algorithm, called *Greedy Best First*, which uses the cost function $f(S) = g(S) + w * h(S)$, where g(S) is the accumulated cost (number of actions when regressing from goal state) and $h(S)$ is the heuristic value of state S.

The heuristic is computed under the assumption that the propositions constituting a state are strictly independent. Thus the cost of a state is estimated as the sum of the cost for each individual proposition making up that state.

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

The heuristic cost $h(p)$ of an individual proposition p is computed using a iterative procedure that is run to fix point as follows. Initially, each proposition p is assigned a cost 0 if it is in the initial state I, and ∞ otherwise. For each instantiated action a, let $Add(a)$, $Del(a)$ and $Prec(a)$ be its Add, Delete and Precondition lists. For each action a that adds some proposition p, $h(p)$ is updated as:

$$h(p) := \min\{h(p), 1 + h(Prec(a))\} \tag{1}$$

Where $h(Prec(a))$ is computed using the sum heuristic (heuristic 1). The updates continue until the h values of all the individual propositions stabilize. This computation can be done before the backward search actually begins, and typically proves to be quite cheap.

Because of the independence assumption, the sum heuristic turns out to be inadmissible (overestimating) when there are positive interactions between subgoals (i.e achieving some subgoal may also help achieving other subgoals), and less informed (significantly underestimating) when there are negative interactions between subgoals (i.e achieving a subgoal deletes other subgoals). Bonet and Geffner [4] provide two separate improvements aimed at handling these problems to a certain extent. Their simple suggestion to make the heuristic admissible is to replace the summation with the "max" function.

Heuristic 2 (Max heuristic) $h_{max}(S) := \max_{p \in S} h(p)$

This heuristic, however, is often much less informed than the sum heuristic as it grossly underestimates the cost of achieving a given state.

To improve the informedness of the sum heuristic, HSP-R adopts the notion of mutex relations first originated in Graphplan planning graph. But unlike Graphplan, only *static propositional mutexes* (also known as binary invariants) are computed. Two propositions p and q form a static mutex when they cannot both be present in any state reachable from the initial state. The static mutex relation can be seen as an extreme form of negative interactions. Since the cost of any set containing a mutex pair is infinite, we define a variation of the sum heuristic called the "sum mutex" heuristic as follows:

Heuristic 3 (Sum Mutex heuristic)
$h(S) := \infty \ if \ \exists_{p,q \in S} \ s.t. \ mutex(p,q) \ else \ \sum_{p \in S} h(p)$

In practice, the Sum Mutex heuristic turns out to be much more powerful than the sum heuristic and HSP-R implementation uses it as the default.

Before closing this section, we provide a brief summary of the procedure of computing mutexes used in HSP-R[4]. The basic idea is to start with a large set of "potential" mutex pairs and iteratively weed out those pairs that cannot be actually mutex. The set M_0 of potential mutexes is union of set M_A of all pairs of propositions $\langle p, q \rangle$, such that for some action a in A, p in $Add(a)$ and q in $Del(a)$, and set M_B of all pairs $\langle r, q \rangle$, such that for some $\langle p, q \rangle$ in M_A and some action a, r in $Prec(a)$ and p in $Add(a)$. This already precludes from consideration potential mutexes $\langle r, s \rangle$, where r and s are not in the add, precondition and delete lists of any single action. As we shall see below, this turns out to be an important limitation in several domains.

2.1 A pathological example that showcases the limitations of sum mutex heuristic

The *sum mutex heuristic* used by HSP-R, while shown to be powerful in domains where the subgoals are relatively independent such as logistics and gripper domains [4], thrashes badly in problems where there is rich interaction between actions and subgoal sequencing. Specifically, when a subgoal that can be achieved early but that must be deleted much later when other subgoals are achieved, the sum heuristic is unable to recognize this interaction. To illustrate this, consider a simple problem from the grid domain [34] shown in Figure 1: Given a 3x3 grid. The initial state is denoted by two propositions at(0,0) and key(0,1) and the goal state is denoted by 2 subgoals at(0,0) and key(2,2) (See figure 1). Notice the subgoal interaction here: The first subgoal at(0,0) is already true in the initial state. When key(2,2) is first achieved, at(0,0) is no longer true and needs to be achieved again. There are three possible actions: the robot moves from one square to an adjacent square, the robot picks up a key if there is such a key in the square the robot currently resides, and the robot drops the key at the current square. One obvious solution is: The robot goes from (0,0) to (0,1), picks up the key at (0,1), moves to (2,2), drops the key there, and finally moves back to (0,0). This is in fact the optimal 10-action plan. We have run (our Lisp implementation of) HSP-R planner on this problem and no solution was found after 1 hour (generating more than 400,000 nodes, excluding those pruned by the mutex computation). The original HSP-R written in C also runs out of memory (250MB) on this problem.

It is easy to see how HSP-R goes wrong. First of all, according to the mutex computation procedure described above, we are able to detect that when the robot is at a square, it cannot be in an adjacent square. But HSP-R's mutex computation cannot detect the type of mutex that says that the robot can also not be in any other square as well (because there is no single action that can place a robot from a square to another square not adjacent to where it currently resides).

Now let's see how this limitation of *sum mutex heuristic* winds up fatally misguiding the search. Given the subgoals (at(0,0), key(2,2)), the search engine has three potential actions over which it can regress the goal state (see Figure 1b). Two of these– move from (0,1) or (1,0) to (0,0)–give the subgoal at(0,0), and the third–dropping key at (2,2), which requires the precondition at(2,2)–gives the subgoal key(2,2). If either of the move actions is selected, then after the regression the robot would be at either (0,1) or (1,0), and

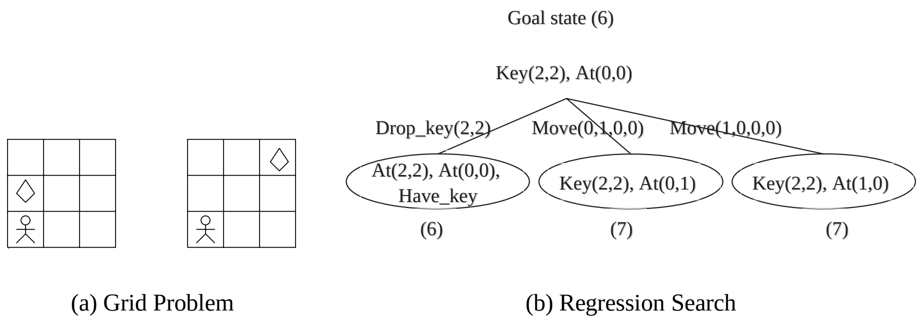


Figure 1: A simple grid problem and the first level of regression search on it.

that would increase the heuristic value because the cost of at(0,1) or at(1,0) is 1 (greater than the cost of at(0,0)). If we pick the dropping action, then after regression, we have a state that has both at(0,0) (the regressed first subgoal), and at(2,2) (the precondition of drop key at (2,2) action). While we can see that this is an inconsistent state, the mutex computation employed by HSP-R does not detect this (as explained above). Moreover, the heuristic value for this invalid state is actually smaller compared to the other two states corresponding to regression over the move actions. This completely misguides the planner into wrong paths, from which it never recovers.

HSP-R also fails or worsens the performance for similar reasons in the travel, mystery, grid, blocks world, and eight puzzle domains[33].

3 Exploiting the structure of Graphplan's planning graph

In the previous section, we showed the type of problems where ignoring the (negative) interaction between subgoals in the heuristic often lead the search into wrong directions. On the other hand, Graphplan's planning graph, with its wealth of mutex constraints, contains much of such information, and can be used to compute more effective heuristics.

Graphplan algorithm [3] works by converting the planning problem specifications into a planning graph. Figure 2 shows part of the planning graph constructed for the 3x3 grid problem shown in Figure 1. 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 stati-

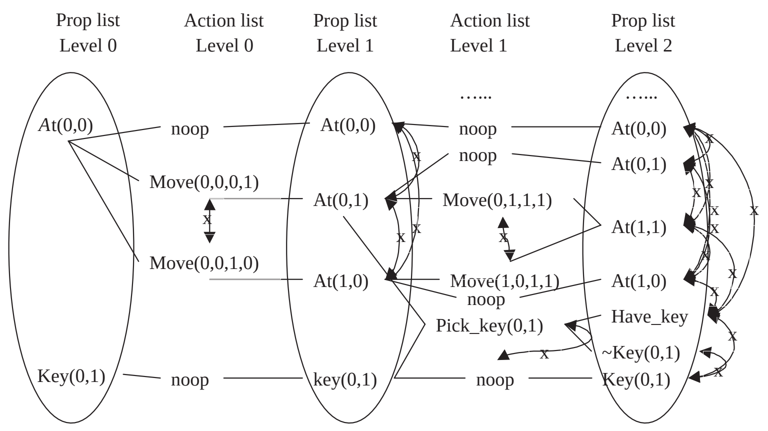


Figure 2: Planning Graph for the 3x3 grid problem

cally 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 2 shows a part of the planning graph for the robot problem specified in Figure 1. The curved lines with x-marks denote the mutex relations. The planning graph can be seen as a CSP encoding [19, 44], with the mutex propagation corresponding to a form of directed partial 1- and 2-consistency enforcement [19]. *Higher order mutexes* can also be computed and propagated in a similar fashion, corresponding to a higher degree of consistency enforcement. The CSP encoding can be solved using any applicable CSP solving methods (a special case of which is the Graphplan's backward search procedure).

Normally, Graphplan attempts to extract a solution from the planning graph of length l, and will expand it to level $l+1$ only if that solution extraction process fails. Graphplan algorithm can thus guarantee that the solution it finds is *optimal* in terms of number of steps. To make the optimality hold in terms of number of actions (a step can have multiple actions), we need to start with a planning graph that is **serial** [17]. A *serial planning graph* is a planning graph in which every pair of non-noop actions at the same level are marked mutex. These additional action mutexes propagate to give additional propositional mutexes. A planning graph is said to **level off** when there is no change in the action, proposition and mutex lists between two consecutive levels.

Based on the above mutex computation and propagation rules, the following properties can be easily verified:

1. The number of actions required to achieve a pair of propositions is no less than the number of proposition levels to be expanded until the two propositions both appear and are *not mutex* in the planning graph.

2. Proposition pairs that remain mutex at the level where the planning graph levels off can never be achieved starting from initial state.

3. The set of actions present in the level where the planning graph levels off contains all actions that are applicable to states reachable from the initial state.

The three observations above give a rough indication as to how the information in the planning graph after it levels off, can be used to guide state search planners. The first observation shows that the level information in planning graph can be used to estimate the cost of achieving a set of propositions. Furthermore, the set of *level-specific* propositional mutexes help give a finer distance estimate. The second observation shows that once the planning graph levels off, all mutexes in the final level are *static* mutexes. The third observation shows a way to extract a finer (smaller) set of applicable actions to be considered by the regression search, since a new action is introduced into a level only if all of its preconditions appear in the previous level and are non-mutexed, and all actions present in a level are also present in the next level.

4 Extracting effective state-space heuristics from planning graph

We describe in this section the formulation of a family of state space heuristics extracted from the planning graph. Briefly, from a graph viewpoint, we will try to estimate the distance from the initial state to a given state (i.e a set of propositions or subgoals). From the viewpoint of action cost, we try to estimate the cost of achieving a given set of subgoals from the initial state.

Before we go on to describing a set of effective heuristics extracted from the planning graph, let us briefly describe how these heuristics are used and evaluated. All the heuristics extracted from the planning graph as well as the HSP-R's sum heuristic are plugged into the *same* regression search engine using a variation of A* search's cost function $f(S) = g(S) + w * h(S)$.

We tested the heuristics on a variety of planning domains. These include several well-known benchmark domains such as the blocksworld, rocket, logistics, 8-puzzle, gripper, mystery, grid and travel. Some of these were used in the AIPS-98 competition [33]. These domains are believed to represent different types of planning difficulty. Problems in the rocket, logistics and gripper domains are typical of those where the subgoals are relatively independent. These domains are also called parallel domains, because of the nature of their solutions. The grid, travel and mystery domains add to logistic domains the hardness of the "topological" combinatorics, while the blocksworld and 8-puzzle domains also have very rich interactions between actions and subgoal sequencing.

This section is concerned with improving the effectiveness of the heuristics and the solution quality without insisting on the strict admissibility of the heuristic functions. We set $w = 1$ in all experimental results described in this subsection, except for the parallel domains (e.g rocket, logistics and gripper) where the heuristics work best (in terms of speed) with $w = 5$. [1] Section 5 is concerned with improving admissible heuristics for finding optimal solutions. Due to the nature of the paper, we will briefly present the empirical results along the formulation of each heuristic, setting motivation for the development of the next heuristics. To make the comparisons meaningful, all the planners are implemented in Allegro Common Lisp, and share most of the critical data structures. The empirical studies are conducted on a 500 MHz Pentium-III with 256 meg RAM, running Linux. All the times reported include both heuristic computation time and search time, unless specified otherwise.

We are now ready to extract heuristics from the planning graph. Unless stated otherwise, we will assume that we have a serial planning graph that has been expanded until it has leveled off (without doing any solution extraction). In this section, we will concentrate on the effectiveness, without insisting on the admissibility of the heuristics.

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 no such level exists, then $lev(S) = \infty$. Similarly, denote $lev(p)$ as the index of the first level that a proposition p comes into the planning graph.*

This property follows immediately from the above definition.

Proposition 1 *If only $binary$ mutexes are present in the planning graph, then:*

$$lev(S) = \max_{p_1, p_2 \in S} lev(\{p_1, p_2\})$$

[1] See [29] for the role of w in BFS. See also [4].

If mutexes up to order k are present in the planning graph, then:

$$lev(S) = \max_{p_1,p_2,...,p_k \in S} lev(\{p_1, p_2, ..., p_k\}) \ for \ |S| \geq k$$

It is easily seen that in order to have increased value of function $lev(S)$, one have to compute and propagate more mutexes, including those of higher-order.

It takes only a small step from the observations made in the previous section to arrive at our first heuristic:

Heuristic 4 (Set-level heuristic) $h_{lev}(S) := lev(S)$

Consider the set-level heuristic in the context of the robot example in previous section. In the planning graph, the subgoal key(2,2) first comes into the planning graph at the level 6, however at that level this subgoal is mutexed with another subgoal at(0,0), and the planning graph has to be expanded 4 more levels until both subgoals are present and non-mutex. Thus the cost estimate yielded by this heuristic is 10, which is exactly the true cost achieving both subgoals.

It is easy to see that set-level heuristic is *admissible*. Secondly, it can be significantly more informed than the *max heuristic*, because the max heuristic is only equivalent to the level that a single proposition first comes into the planning graph. Thirdly, a by-product of the set-level heuristic is that it already subsumes much of the static mutex information used by the Sum Mutex heuristic. Moreover, the propagated mutexes in the planning graph wind up being more effective in detecting static mutexes that are missed by HSP-R. In the context of the robot example, HSP-R can only detect that a robot cannot be at squares adjacent to its current square, but using planning graph, we are able to detect that the robot cannot be at any square other than its current square.

Table 1 and 2 show that the set-level heuristic performs reasonably well in domains such as grid, mystery, travel and 8-puzzle[2] compared to the standard Graphplan.[3] Many of these problems prove intractable for HSP-R's sum-mutex heuristic. We attribute this performance of the set-level heuristic to the way the negative interactions between subgoals are accounted for by the level information.

Interestingly, the set-level heuristic fails in the domains that the *sum heuristic* typically does well, such as rocket world and logistics, where the subgoals are fairly independent of each other. Closely examining the heuristic values reveals that the set-level heuristic remains too conservative and often underestimates the real cost in these domains. A related problem is that the range of numbers that the cost of a set of propositions can take is limited to integers less than or equal to the length of the planning graph. This range limitation leads to a practical problem as these heuristics tend to attach the same numerical cost to many qualitatively distinct states, forcing the search to resort to arbitrary tie breaking.

To overcome these limitations, we pursue two families of heuristics that try to account for the subgoal interactions in different ways. The first family, called "partition-k" heuristics, attempts to improve and generalize the set-level heuristic using the sum heuristic's idea. Specifically, it estimates the cost of a set in terms of costs of its partitions. The second family, called "adjusted sum" heuristics attempt to improve the sum heuristic using the set-level's idea. Specifically, it starts explicitly from the sum heuristic and then considers adding the interactions among subgoals that can be extracted from the planning graph's level information. These two families are described separately in the next two subsections.

[2]8puzzle-1, 8puzzle-2 and 8puzzle-3 are two hard and one easy eight puzzle problems of solution length 31, 30 and 20, respectively. Grid3 and grid4 are simplified from the grid problem at AIPS-98 competitions by reducing number of keys and grid's size.

[3]Graphplan implemented in Lisp by M. Peot and D. Smith.

Problem	Graphplan	Sum-mutex	set-lev	partition-1	partition-2	adj-sum	combo	adj-sum2
bw-large-a	12/ 13.66	12/ 41.64	12/ 26.33	12/ 19.02	14/ 19.79	12/ 17.93	12/ 20.38	12/ 19.56
bw-large-b	18/ 379.25	18/ 132.50	18/ 10735.48	18/ 205.07	20/ 90.23	22/ 65.62	22/ 63.57	18/ 87.11
bw-large-c	-	-	-	-	32/ 535.94	30/ 724.63	30/ 444.79	28/ 738.00
bw-large-d	-	-	-	-	48/ 2423.32	-	-	36/ 2350.71
rocket-ext-a	-	36/ 40.08	-	32/ 4.04	54/ 468.20	40/ 6.10	34/ 4.72	31/ 43.63
rocket-ext-b	-	34/ 39.61	-	32/ 4.93	34/ 24.88	36/ 14.13	32/ 7.38	28/ 554.78
att-log-a	-	69/ 42.16	-	65/ 10.13	66/ 116.88	63/ 16.97	65/ 11.96	56/36.71
att-log-b	-	67/ 56.08	-	69/ 20.05	66/ 113.42	67/ 32.73	67/ 19.04	47/ 53.28
gripper-15	-	45/ 35.29	-	45/ 12.55	45/ 178.39	45/ 16.94	45/ 16.98	45/ 14.08
gripper-20	-	59/ 90.68	-	59/ 39.17	61/ 848.78	59/ 20.54	59/ 20.92	59/38.18
8-puzzle1	31/ 2444.22	33/ 196.73	31/ 4658.87	35/ 80.05	39/ 130.44	39/ 78.36	39/ 119.54	31/ 143.75
8-puzzle2	30/ 1545.66	42/224.15	30/ 2411.21	38/ 96.50	36/ 145.29	42/ 103.70	48/ 50.45	30/ 348.27
8-puzzle3	20/ 50.56	20/ 202.54	20/ 68.32	20/ 45.50	20/ 232.01	24/ 77.39	20/ 63.23	20/ 62.56
travel-1	9/ 0.32	9/ 5.24	9/ 0.48	9/ 0.53	9/ 0.77	9/ 0.42	9/ 0.44	9/ 0.53
grid3	16/ 3.74	-	16/ 14.09	16/ 55.40	16/ 121.94	18/ 21.45	19/ 18.82	16/ 15.12
grid4	18/ 21.30	-	18/ 32.26	18/ 86.17	18/ 1261.66	18/ 37.01	18/ 37.12	18/ 30.47
aips-grid1	14/ 311.97	-	14/ 659.81	14/ 870.02	14/ 1142.83	14/ 679.36	14/ 640.47	14/ 739.43
mprime-1	4/ 17.48	-	4/ 743.66	4/ 78.730	4/ 565.47	4/ 76.98	4/ 79.55	4/ 722.55

Table 1: Number of actions/ Total CPU Time in seconds. The dash (-) indicates that no solution was found in 3 hours or 500MB. All the heuristics are implemented in Lisp and share the same state search data structure and algorithm. The empirical studies are conducted on a 500 MHz Pentium-III with 512 Meg RAM, running Linux.

Problem	Graphplan	Sum-mutex	set-lev	partition-1	partition-2	adj-sum	combo	adj-sum2
bw-large-a	-	77/ 12	456/ 108	71/ 13	101/ 19	71/ 13	66/ 12	83/ 16
bw-large-b	-	210/ 34	315061/ 68452	15812/ 3662	1088/ 255	271/ 56	171/ 37	1777/ 338
bw-large-c	-	-	-	-	1597/ 283	8224/ 1678	747/ 142	8248/ 1399
bw-large-d	-	-	-	-	5328/ 925	-	-	10249/ 1520
rocket-ext-a	-	769/ 82	-	431/ 43	16246/ 3279	658/ 69	464/ 41	3652/ 689
rocket-ext-b	-	633/ 66	-	569/ 50	693/ 64	1800/ 173	815/ 72	60788/ 8211
att-log-a	-	2978/ 227	-	1208/ 89	3104/ 246	2224/ 157	1159/ 85	1109/ 74
att-log-b	-	3405/ 289	-	1919/ 131	2298/ 161	4219/ 289	1669/ 115	1276/ 85
gripper-15	-	1775/ 178	-	1775/ 178	4204/ 1396	1358/ 150	1350/ 150	503/ 95
gripper-20	-	3411/ 268	-	3411/ 263	10597/ 3509	840/ 149	840/ 149	846/ 152
8-puzzle1	-	1078/ 603	99983/ 56922	1343/ 776	1114/ 613	1079/ 603	1980/ 1129	2268/ 1337
8-puzzle2	-	1399/ 828	51561/ 29176	1575/ 926	1252/ 686	1540/ 899	475/ 279	6544/ 3754
8-puzzle3	-	2899/ 1578	1047/ 617	575/ 318	2735/ 1539	1384/ 749	909/ 498	765/ 440
travel-1	-	4122/ 1444	25/ 9	93/ 59	97/ 60	40/ 18	40/ 18	37/ 18
grid3	-	-	49/ 16	3222/ 1027	4753/ 1443	1151/ 374	865/ 270	206/ 53
grid4	-	-	44/ 18	7758/ 4985	24457/ 14500	1148/ 549	1148/ 549	51/ 18
aips-grid1	-	-	108/ 16	5089/ 864	9522/ 1686	835/ 123	966/ 142	194/ 25

Table 2: Number of nodes generated/ expanded. The dash (-) indicates that no solution was found after generating more than 500000 nodes, or runs out of memory.

4.1 Partition-k heuristics

When the subgoals are relatively independent, the summation of the cost of each individual gives a much better estimate, whereas the graph level value of the set tends to underestimate significantly. To avoid this problem and at the same time keep track of the interaction between subgoals, we want to partition the set S of propositions into subsets, each of which has k elements: $S = S_1 \cup S_2 ... \cup S_m$ (if k does not divide $|S|$, one subset will have less than k elements), and then apply the set-level heuristic value on each partition. Ideally, we want a partitioning such that elements within each subset S_i may be interacting with each other, but the subsets are independent (i.e non-interacting) of each other. By *interacting* we mean the two propositions form either a pair of dynamic (level-specific) or static mutex in the planning graph. These notions are formalized by the following definitions.

Definition 2 *The* **(binary) interaction degree** *between two propositions* p_1 *and* p_2 *is defined as* $\delta(p_1, p_2) = lev(\{p_1, p_2\}) - max\{lev(p_1), lev(p_2)\}$.

When $p1$ and $p2$ are dynamic mutex, $\delta(p_1, p_2) = lev(\{p_1, p_2\}) - max\{lev(p_1), lev(p_2)\} > 0$ but $lev(\{p_1, p_2\}) < \infty$. When $p1$ and $p2$ are static mutex, $lev(\{p_1, p_2\}) = \infty$. Since we only consider propositions that are present in the planning graph, i.e $max\{lev(p_1), lev(p_2)\} < \infty$, it follows that $\delta(p_1, p_2) > 0$ as well. When $p1$ and $p2$ are neither type of mutex, $lev(\{p_1, p_2\}) = max\{lev(p_1), lev(p_2)\}$, thus $\delta(p_1, p_2) = 0$.

Definition 3 *Two propositions* p *and* q *are* **not interacting** *with each other if and only if* $\delta(p, q) = 0$. *Two sets of propositions* S_1 *and* S_2 *are not interacting with each other if no proposition in* S_1 *is interacting with a proposition in* S_2.

We are now ready to state the family of partitioning heuristics:

Heuristic 5 (Partition-k heuristic)
$h_{part-k}(S) := \sum_{S_i} lev(S_i)$, *where* $S_1, ..., S_m$ *are k-sized partitions of* S.

The question of deciding the partioning parameter k, and how to partition the set S when $1 < k < |S|$, however, is interesting. We find out that this knowledge may be largely domain-dependent. For example, for $k = 1$, we have $h(S) = \sum_{p \in S} lev(p)$. The *partition-1* heuristic exhibits similar behavior compared to *sum-mutex* heuristic in domains where the subgoals are fairly independent (e.g gripper, logistics, rocket), and it is clearly better than sum-mutex in all other domains except the blocks world (see table 1).

For $k = |S|$, we have the *set-level* heuristic, which is very good in a complementary set of domains, compared with the sum-mutex heuristic.

For $k = 2$, we implemented a simple pairwise partitioning scheme as follows. For each proposition p_1 in a given set S, we choose the proposition p_2 that gives the greatest interaction degree $\delta(p_1, p_2)$ with p_1 to make up a pairwise partition $\{p_1, p_2\}$ (breaking ties arbitrarily).

As Table 1 shows, the resulting heuristic exhibits interesting behavior: It can solve many problems that are either intractable by the *sum heuristic* or the *set-level heuristic*. In fact, the *partition-2* heuristic can scale up very well in domains such as the blocks world, returning a 48 steps solution for bw-large-d (19 blocks) after generating just 5328 nodes and expanding 925 nodes in 2423 sec.

While it is not likely that a single partitioning scheme will be effective across different domains, it would be interesting to have a fuller account of behavior of the family of *partition-k heuristics*, depending on the partitioning parameters, with respect to different problem domains.

Another attractive idea is to consider "adaptive partition" heuristics that do not insist on equal sized partitions. Based on the above definition of subgoal interaction, we attempt a simple procedure that we call

adaptive partitioning of a given set S as follows. Given a set S, choose any proposition $p_1 \in S$. Initially, let $S_1 = \{p_1\}$. For each proposition $p \in S_1$, add to set S_1 all propositions in $S - S_1$ that are interacting with p. This is repeated until S_1 becomes unchanged. Thus we have partitioned S into S_1 and $S - S_1$, where these two subsets are not interacting with each other whereas each proposition in S_1 interacts with some other member in the same set. This procedure is recursively applied on the set $S - S_1$ and so on.

Unfortunately, a partitioning heuristic that uses this adaptive partitioning and then applies the set-level heuristic on each partitions does not appear to scale up well in most domains that we have tested. The reason, we think, is that since the set of mutex constraints that we have been able to exploit from the planning graph so far involve only two state variables (i.e binary mutex), it may not be as helpful to apply the set-level heuristic on partitions of size greater than two.

4.2 Adjusted Sum Heuristics

We now consider improving the sum heuristic by considering both negative and positive interactions among propositions. Since fully accounting for either type of interactions alone can be as hard as the planning problem itself, we circumvent this difficulty by using partial relaxation assumption on the subgoal interactions. Namely, we ignore one type of subgoal interactions in order to account for the other, and then combine them both together.

First of all, it is simple to embed the sum heuristic value into the planning graph. We maintain a cost value for each new proposition. Whenever a new action is introduced into the planning graph, we update the value for that proposition using the same updating rule 1 in Section 2. Denote $cost(p)$ as the cost of achieving a proposition p according to the sum heuristic.

We are now interested in estimating the cost $cost(S)$ for achieving a set $S = \{p_1, p_2, ..., p_n\}$. Suppose $lev(p_1) \leq lev(p_2) \leq ... \leq lev(p_n)$. Under the assumption that all propositions are independent, we have

$$cost(S) := cost(S - p_1) + cost(p_1)$$

Since $lev(p_1) \leq lev(S - p_1)$, proposition p_1 is *possibly* achieved before the set $S - p_1$.[4] Now, we assume that there are no positive interactions, but there are negative interactions between the propositions. Therefore, upon achieving $S - p_1$, subgoal p_1 may have been deleted and needs to be achieved again. This information can be extracted from the planning graph. According to the planning graph, set $S - p_1$ and S are *possibly* achieved at level $lev(S - p_1)$ and level $lev(S)$, respectively. If $lev(S - p_1) \neq lev(S)$ that means there is some interaction between achieving $S - p_1$ and achieving p_1, because the planning graph has to expand up to $lev(S)$ to achieve both $S - p_1$ and p_1. To take this negative interaction into account, we assign:

$$cost(S) := cost(S - p_1) + cost(p_1) + (lev(S) - lev(S - p_1)) \tag{2}$$

Applying this formula to $S - p_1, S - p_1 - p_2$ and so on, we derive:

$$cost(S) := \sum_{p_i \in S} cost(p_i) + lev(S) - lev(p_n)$$

Since $lev(p_n) = \max_{p_i \in S} lev(p_i)$ as per our setup, we have the following heuristic:

Heuristic 6 (Adjusted-sum heuristic)
$$h_{adjsum}(S) := \underbrace{\sum_{p_i \in S} cost(p_i)}_{h_{sum}(S)} + \underbrace{lev(S)}_{h_{lev}(S)} - \underbrace{\max_{p_i \in S} lev(p_i)}_{\approx h_{max}(S)}$$

[4]For brevity, we use the notation $S - p_1$ to mean $S - \{p_1\}$.

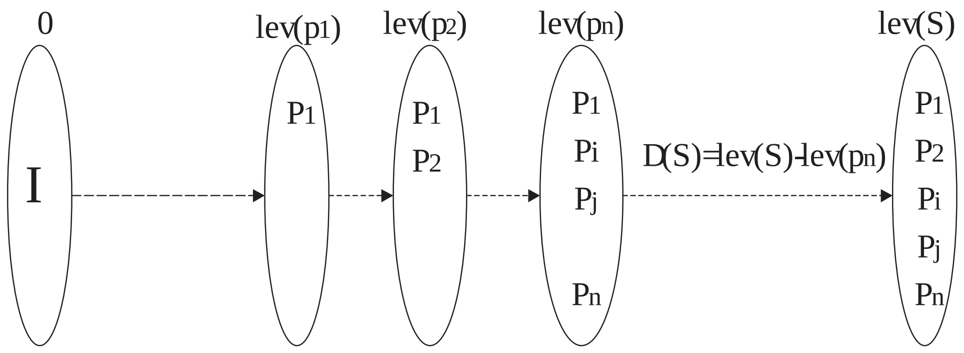


Figure 3: The cost incurred for ignoring the negative interactions.

Table 1 and 2 show that this heuristic does very well across *all* different types of problems that we have considered. To understand the robustness of the heuristic, notice that the first term in its formula is exactly the *sum* heuristic value, while the second term is the *set-level heuristic*, and the third *approximately* the *max* heuristic.

Definition 4 *The* **interaction degree** *among propositions in a set S is captured by*
$\Delta(S) = lev(S) - \max_{p \in S} lev(p)$

Therefore, the adjusted-sum heuristic function can be written as:

$$h_{adjsum}(S) = h_{sum}(S) + \Delta(S) \tag{3}$$

The heuristic function $h_{adjsum}(S)$ can be seen as being composed of two different component: $h_{sum}(S)$ is the estimated cost of achieving S under the *independence* assumption, while $\Delta(S)$ accounts for the additional cost incurred by the *negative* interactions.

Notice that the notion of *(binary) interaction degree* between two propositions $\delta(p_1, p_2)$ is obviously a special case of the above definition when $|S| = 2$, i.e, $\delta(p_1, p_2) = \Delta(\{p_1, p_2\})$. It is simple to see that when there are no negative interactions among propositions, $\Delta(S) = 0$, because

$$h_{lev}(S) = lev(S) = \max_{p \in S} lev(p) \approx h_{max}(S)$$

From table 1 the solutions provided by adjusted sum heuristic are longer than those provided by other heuristics in many problems. The reason for this is that the first term $h_{sum}(S) = \sum_{p_i \in S} cost(p_i)$ actually overestimates, because in many domains achieving some subgoal typically also helps achieve others. We are interested in improving the *adjusted-sum* heuristic by replacing the first term in its formula by another estimation $cost_p(S)$ that takes into account this type of *positive* interactions while ignoring the negative interactions (which are anyway accounted for by $\Delta(S)$).

Since we are ignoring the negative interactions, once a subgoal is achieved, it will never be deleted again. Furthermore, the order of achievement of the subgoals $p_i \in S$ would be roughly in the order of $lev(p_i)$. Let p_S be the proposition in S such that $lev(p_S) = \max_{p_i \in S} lev(p_i)$. p_S will *possibly* be the last proposition that is achieved in S. Let a_S be an action in the planning graph that achieves p_S in the level $lev(p_S)$, where p_S first appears. (If there is more than one such action, none of them would be noop actions, and we would select one randomly.)

By regressing S over action a_S, we have state $S + Prec(a_S) - Add(a_S)$. Thus, we have the recurrence relation (assuming unit cost for the selected action a_S)

$$cost_p(S) := 1 + cost_p(S + Prec(a_S) - Add(a_S)) \tag{4}$$

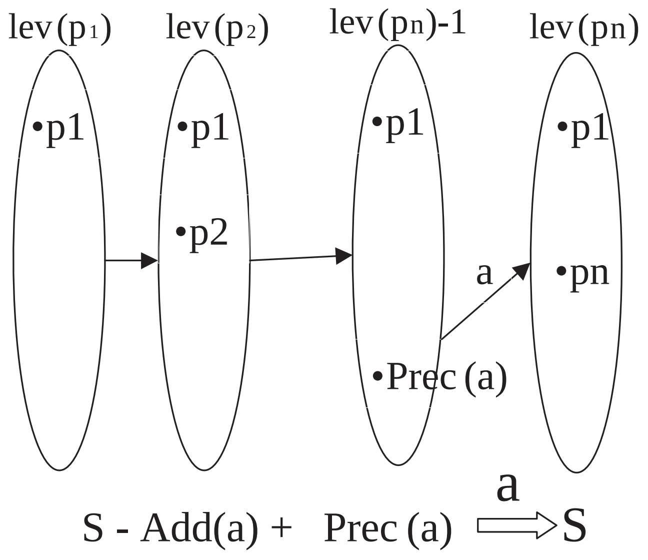


Figure 4: Extracting the length of a plan that ignores the negative interactions.

The positive interactions are accounted for by this regression in the sense that by subtracting $Add(a_S)$ from S, any proposition that is co-achieved when p_S is achieved is not counted in the cost computation. Since $lev(Prec(a_S))$ is strictly smaller than $lev(p_S)$, recursively applying equation 3 to its right hand side will eventually reduce to state S_0 where $lev(S_0) = 0$, and $cost_p(S_0)$ is 0.

It is interesting to note that the repeated reductions involved in computing $cost_p(S)$ indirectly extract a sequence of actions (the a_S selected at each reduction), which would have achieved the set S from the initial state if there were no negative interactions. In this sense, $cost_p(S)$ is similar in spirit to the "relaxed plan" heuristic recently proposed by Hoffman[15].

Replacing $h_{sum}(S)$ with $cost_p(S)$ in the definition of h_{adjsum}, we get an improved version of adjusted sum heuristic that takes into account both positive and negative interactions among propositions.

Heuristic 7 (Adjusted-sum2 heuristic) $h_{adjsum2}(S) := cost_p(S) + \Delta(S)$
where $cost_p(S)$ is computed using equation 4, and $\Delta(S)$ is defined by equation 4.

Table 1 shows that adjusted-sum2 heuristic can solve all types of problems considered. The heuristic is only slightly worse compared to the adjusted-sum heuristic in term of speed, but gives a much better solution quality. In our experiments, with the exception of problems in the rocket domains, the adjusted-sum2 heuristic value is usually admissible and often gives optimal or near optimal solutions.

Finally, another way of viewing the adjusted-sum heuristic is that, it is composed of $h_{sum}(S)$, which is good in domains where subgoals are fairly independent, and $h_{lev}(S)$, which is good in a complementary set of domains (see table 1). Thus the summation of them may yield a combination of *differential* power effective in wider range of problems, while discarding the third term $h_{max}(S)$ may sacrifice the solution quality.

Heuristic 8 (Combo heuristic) $h(S) := h_{sum}(S) + h_{lev}(S)$, *where $h_{sum}(S)$ is the sum heuristic value and $h_{lev}(S)$ is the set-level heuristic value.*

Surprisingly, as shown in table 1 the Combo heuristic is even slightly faster than adjusted-sum heuristic across all types of problems while the solution quality remains comparable.

Since the family of adjusted-sum heuristics turns out to be the most effective, we devote the next subsection to further analysis of and possible avenues of improvement for this heuristic family.

4.3 Improvements of the adjusted-sum heuristic functions

So far, we have started from the sum heuristic and tried to improve it by accounting for both negative and positive subgoal interactions extracted from the planning graph. The formulation, culminated in the derivation of *adjusted-sum2* heuristic, is quite interesting in the way it supposedly accounts for subgoal interactions. This heuristic function is composed of two components: $cost_p(S)$, which estimates the cost of the plan by trying to accounting for the positive interactions while ignoring the negative interactions, and $\Delta(S)$, which intends to measure the "penalty" cost for ignoring such negative interactions. In this subsection, we will discuss several ways in which this type of heuristic functions can be further improved. Naturally, it involves improving the component $cost_p(S)$ as well as $\Delta(S)$.

The first component, $cost_p(S)$, returns the cost of a plan that solves the relaxed version of the problem in which the negative effects of actions are completely ignored. Although the positive interactions are accounted for in its computation, $cost_p(S)$ is by no means guaranteed to be a lower bound of the optimal solution the original problem.[5] The reason for this is that there may be more than one plan satisfying the relaxed problem. Taking the minimum-length plan among *all* possible solutions to the relaxed problem would ensure a lower bound of the original problem's optimal plan. However, this task is known to be NP-hard. We can see that improving the admissibility of the heuristic function $h_{adjsum2}(S)$ directly involves improving the admissibility of the component $cost_p(S)$, i.e lowering its value. Thus, one way to do this would be considering more than one plan for the relaxed problem and then taking the minimum plan cost. This will make the computation more costly.

The second component in adjusted-sum2 heuristic function $h_{adjsum2}(S)$, $\Delta(S) = lev(S) - \max_{p_i \in S} lev(p_i)$, accounts for the negative interactions among subgoals. Improving the informedness of the heuristic function involves a more accurate account of the negative interactions captured by $\Delta(S)$, i.e increasing its value. One way to do this is to improve the accuracy of $lev(S)$. Recall that the $lev(S)$, by definition, is the index of the first level in the planning graph that all propositions in S are present and not mutex with one another. Following the proposition 1, improving the accuracy of $lev(S)$ involves computing and propagating more mutexes, including higher order ones. Unfortunately, this task is also computationally intensive.[6] In section 5 we shall investigate this in detail in the quest to improve the *admissible* set-level heuristic for efficient *optimal* planning.

For now, let us consider improving $\Delta(S)$ given only binary mutex constraints. In fact, the effectiveness of the adjusted-sum2 heuristic, evidenced by the very low number of nodes the search needs to expand to reach an *approximately* optimal solution, shows that binary mutex constraints are often quite sufficient to account for subgoal interactions in most typical planning domains of reasonable sizes. The challenge is how to exploit this set of constraints most effectively. The answer may, again, depends on the nature of each specific domain in question.

Note that the term $\Delta(S) = lev(S) - \max_{p_i \in S} lev(p_i)$ gives an intuitive measure for the cost incurred by the negative interactions, in the sense that at least as many actions are needed to account for the interactions. Since we are given only the set of *binary* mutex constraints, i.e $lev(S)$ is simply computed as $lev(S) = \max_{p_1, p_2 \in S} lev(\{p_1, p_2\})$, according to proposition 1. In this case, we may want to improve the negative penalty cost function $\Delta(S)$ by accounting for the interactions more aggressively. Specifically, we might explicitly consider the interactions between every *pair* of individual propositions in a given set $S = \{p_1, p_2, ..., p_n\}$. For each pair of p and q, the interaction is reflected by its *interaction degree*, as de-

[5]Although, in practice, $cost_p(S)$ often returns the cost estimate that is admissible.

[6]Even computing all binary mutexes can be as hard as the planning problem [3].

fined in definition 2 as $\delta(p,q) = lev(\{p,q\}) - \max\{lev(p), lev(q)\}$. A conservative estimation for the total cost incurred by negative interactions would be:

$$\Delta_{max}(S) = \max_{p,q \in S} \delta(p,q) \tag{5}$$

Heuristic 9 (Adjsum2M heuristic) $h_{adjsum2M}(S) := cost_p(S) + \Delta_{max}(S)$
where $cost_p(S)$ is computed using equation 4, and $\Delta(S)$ is as given in Equation 5.

It is simple to see that $\Delta_{max}(S) \geq \Delta(S)$. [7] In addition, when there are no interactions among any pair of propositions, $\Delta_{max}(S) = \Delta(S) = 0$. In table 3 we compare the relative effectiveness of adjusted-sum2 and adjusted-sum2M heuristics using Δ and Δ_{max}, respectively. It is interesting to note that both heuristic functions tend to give admissible estimates in most problems. Furthermore, since $\Delta_{max}(S) \geq \Delta(S)$ for all S, $h_{adjsum2M}$ tend to give closer cost estimate than $h_{adjsum2}$, resulting in overall improvement in most problems considered. In particular, the lower number of nodes expanded and generated in the rocket domain is quite significant.

In the domains where the *interactions* themselves are relatively **independent** to one another, the interactions among subgoals might be better captured by summing the interaction degree of all pairs of the propositions. To avoid overcounting such interactions too much, we might want to partition the set S into pairwise subsets $S_i = \{p_{i1}, p_{i2}\}$ and summing the interaction degree of all these pairs, i.e $\delta(p_{i1}, p_{i2}) = \Delta(S_i)$. Thus we would have:

$$\Delta_{sum}(S) = \sum_i \Delta(S_i)$$

In our experiment however, the adjusted-sum2 heuristic using Δ_{sum} does not result in any significant improvement over adjsum2M heuristic on most problems. It is actually worse in several domains. The reason, we think, is that in most domains, the *interactions* themselves tend to interact among one another in complex ways that make it unreasonable to assume that they are independent.

5 Finding optimal plans with admissible heuristics

We now focus on admissible heuristics that can be used to produce optimal plans. While works in the AI search literature concentrate entirely on admissible heuristics (and finding optimal solutions), traditionally, efficient generation of optimal plans has received little attention in the planning community. In [17] Kambhampati *et. al.* point out that Graphplan algorithm is guaranteed to find optimal plans when the planning graph is serial. In contrast, none of the known efficient state space planners [34, 5, 4, 38] can guarantee optimal solutions.[8]

In fact, it is very hard to find an admissible heuristic that is effective enough to be useful across different planning domains. As mentioned earlier, in [5], Bonet et al. introduced the *max heuristic* that is admissible. In the previous section, we introduced the *set-level* heuristic that is admissible and gives closer cost estimate than the max heuristic. We tested the set-level heuristic on a variety of domains using A* search's cost function $f(S) = g(S) + h(S)$. The results are shown in table 4, and clearly establish that set-level heuristic is significantly more effective than max heuristic. Grid, travel and mprime are domains where the set-level heuristic gives very close estimates (see table 1). Optimal search is less effective in domains such as the 8-puzzle and blocks world problem. Domains such as logistics and gripper remain intractable under reasonable limits in time and memory.

[7]Indeed, suppose p_m and p_r are two propositions such that $lev(\{p_m, p_r\}) = \max_{p_i,p_j \in S} lev(\{p_i, p_j\})$. Since we only consider binary mutexes, $\max_{p_i,p_j \in S} lev(\{p_i, p_j\}) = lev(S)$. Thus, $\Delta_{max}(S) \geq \delta(p_m, p_r) = lev(\{p_m, p_r\}) - \max\{lev(p_m), lev(p_r)\} = lev(S) - \max\{lev(p_m), lev(p_r)\} \geq lev(S) - \max_{p_i \in S} lev(p_i) = \Delta(S)$.

[8]But see also a recent work by Haslum & Geffner[14].

Problem	Opt	$h(S) = cost_p(S) + \Delta(S)$			$h(S) = cost_p(S) + \Delta_{max}(S)$		
		Est	Length/Time	#N-gen/#N-exp	Est	Length/Time	#N-gen/#N-exp
bw-large-a	12	13	12/ 19.56	83/ 16	14	12/ 19.96	71/ 13
bw-large-b	18	17	18/ 87.11	1777/ 338	18	18/ 82.56	489/ 95
bw-large-c	28	26	28/ 738.00	8248/ 1399	27	28/ 851.68	6596/ 1131
bw-large-d	36	35	36/ 2350.71	10249/ 1520	35	36/ 2934.11	8982/ 1310
rocket-ext-a	26	26	31/ 43.63	3652/ 689	26	29/ 21.95	438/ 43
rocket-ext-b	25	26	28/ 554.78	60788/ 8211	26	26/ 21.75	401/ 37
att-log-a	52	50	56/ 36.71	1109/ 74	50	55/ 79.22	1069/ 69
att-log-b	44	40	47/ 53.28	1276/ 85	40	45/ 105.09	1206/ 78
gripper-15	45	32	45/ 14.08	503/ 95	32	45/ 33.64	501/ 94
gripper-20	59	42	59/ 38.18	846/ 152	42	59/ 106.80	840/ 149
8puzzle-1	31	26	31/ 143.75	2268/ 1337	28	31/ 659.55	5498/ 3171
8puzzle-2	30	22	30/ 348.27	6544/ 3754	24	30/ 561.80	4896/ 2822
8puzzle-3	20	17	20/ 62.56	765/ 440	19	20/ 67.46	479/ 272
travel-1	9	9	9/ 0.53	37/ 18	9	9/ 0.57	37/ 18
grid3	16	13	16/ 15.12	206/ 53	13	16/ 20.73	382/ 97
grid4	18	18	18/ 30.47	51/ 18	18	18/ 32.80	51/ 18

Table 3: Evaluating the relative effectiveness of adjusted-sum2 heuristic using Δ and Δ_{max} to account for negative interactions. Column Opt shows the length of the optimal sequential plan. Column Est shows the estimated distance from goal to the initial state according to the heuristic. Length/Time shows the length of found plan and total run time in seconds. #N-gen and #N-exp are number of nodes generated and expanded during the search, respectively.

The main problem once again is that the set-level heuristic still hugely underestimates the cost of a set of propositions. The reason for this is that there are many *n-ary* $(n > 2)$ *level-specific* mutex constraints present in the planning graph, that are never marked during planning graph construction, and thus cannot be used by set-level heuristic. This suggests that identifying and using higher order mutexes can improve the effectiveness of the set-level heuristic. Suppose that we consider computing and propagating mutex constraints up to order *k-ary* in the planning graph. The mutex computation and propagation corresponds to computing the set-level value $lev(S)$ for $|S| \leq k$. For $|S| > k$, the set-level heuristic value becomes:

$$h(S) := lev(S) = \max_{p_1, p_2, ..., p_k \in S} lev(\{p_1, p_2, ..., p_k\}) \tag{6}$$

We have presented a family of admissible *set-level* heuristics, whose informedness can be improved by increasing the size (order) of mutexes one wants to utilize. In practice, propagating all higher order mutexes is likely to be an infeasible idea [3, 17], as it essentially amounts to full consistency enforcement of the underlying CSP. According to Blum, even computing and propagating 3-ary mutexes can be very expensive. A seemingly zanier idea is to use a limited run of Graphplan's own backward search, armed with EBL [19], to detect higher order mutexes in the form of "memos". Detecting that a set S of propositions is a memo at level l basically means that $lev(S) \geq l + 1$. Computing this information is less costly that computing the "exact" value of $lev(S)$. On the other hand, knowing the lower bound of $lev(S)$ may often be enough to avoid (or delay) branching on node S. Thus the rationale behind this idea is that we only search for mutexes of different orders (binary and higher) within a limited number of levels without losing much useful information one can provide for the set-level heuristic by full-blown computation of higher-order mutexes. Even within each level, such amount of search involved is limited to control the cost of heuristic computation.

We have implemented this idea by restricting the backward search to a limited number of backtracks $lim = 1000$. This lim can be increased by a factor $\mu > 1$ as we expand the planning graph to next

Problem	Len	max		set-level		w/ memo		GP
		Est	Time	Est	Time	Est	Time	
8puzzle-1	31		-	14	4658	28	1801	2444
8puzzle-2	30	10	-	12	2411	28	891	1545
8puzzle-3	20	8	144	10	68	19	50	50
bw-large-a	12	6	34	8	21	12	16	14
bw-large-b	18	8	-	10	10735	16	1818	433
bw-large-c	28	12	-	14	-	20	-	-
grid3	16	16	13	16	13	16	5	4
grid4	18	10	33	18	30	18	22	22
rocket-ext-a	-	5	-	6	-	11	-	-

Table 4: Column titled "Len" shows the length of the found optimal plan (in number of actions). Column titled "Est" shows the heuristic value the distance from the initial state to the goal state. Column titled "Time" shows CPU time in seconds. "GP" shows the CPU time for **Serial Graphplan**

level. Table 4 shows the performance of the set-level heuristic using a planning graph adorned with learned memos. We note that the heuristic value (of the goal state) as computed by this heuristic is significantly better than the set-level heuristic operating on the vanilla planning graph. For example in 8-puzzle2, the normal set-lev heuristic estimates the cost to achieve the goal as 12, while using memos pushes the cost to 28, which is quite close to the true optimal value of 30. This improved informedness results in a speedup in all problems we considered (up to 3x in the 8-puzzle2, 6x in bw-large-b), even after adding the time for memo computation using limited backward search.

We also compared the performance of the two set-level heuristics with the serial Graphplan, which also produces optimal plans. The set-level heuristic is better in the 8-puzzle problems, but not as good in the blocks world problems (See table 4). We attribute this difference to the fact that more useful mutexes are probably captured by the limited search of memos in the 8-puzzle problems than in the blocks world problems. To capture more useful memos, one could increase the amount of backward search involved.

6 Implementational issues in extracting heuristics from planning graphs

6.1 Improving the efficiency of the heuristic computation

The main cost of heuristic computation lies in the construction of a leveled-off serial planning graph. Once this planning graph is constructed, the useful information necessary for computing a heuristic value of a given set can be accessed quickly from the graph data structure.

Although the planning graph structure is in principle polynomial in terms of time and memory, the cost of its computation in fact varies according to different domains. In the parallel domains where there is little interactions among subgoals, the planning graph consumes very little memory and building time. For example, it takes only seconds to build a planning graph for the att-log-a problem, which has a solution of length 56. On the other hand, in domains where there are strong interactions among subgoals, the resulting planning graph can be very big and costly to build, for having to store many more mutex constraints. The blocks world or grid world are examples of these domains. For instance, the planning graph for bw-large-c takes about 400 seconds to build out of 535 seconds of total run time for partition-2 heuristic (in Lisp). The planning graph may also grow so big that it takes all the memory available.

There are a variety of techniques for improving the efficiency of planning graph construction in terms of both time and space, including bi-level representations that exploit the structural redundancy in the planning graph in STAN planner[31], as well as (ir)relevance detection techniques such as RIFO [35] that ignore ir-

Problem	Normal PG	Bi-level PG	Speedup
bw-large-b	63.57	20.05	3x
bw-large-c	444.79	114.88	4x
bw-large-d	-	11442.14	100x
rocket-ext-a	4.72	1.26	4x
rocket-ext-b	7.38	1.65	4x
att-log-a	11.96	2.27	5x
att-log-b	11.09	3.58	3x
gripper-20	20.92	7.26	3x
8puzzle-1	119.54	20.20	6x
8puzzle-2	50.45	7.42	7x
8puzzle-3	63.23	10.95	6x
travel-1	0.44	0.12	4x
grid-3	18.82	3.04	6x
grid-4	37.12	14.15	3x
aips-grid-1	640.47	163.01	4x
mprime-1	79.55	67.75	1x

Table 5: Total CPU time improvement from efficient heuristic computation for **Combo** heuristic

relevant literals and actions while constructing the planning graph. These techniques can be used to improve the cost of our heuristic computation. In fact, in one of our recent experiments, we have used a bi-level planning graph as a basis for our heuristics.[9] Our experiment results show significant speedups (up to 7x) in all problems, and we are also able to solve more problems than before because our planning graph takes less memory (described in table 5).

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 the state search does not operate 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 [35] 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.

We have implemented the first technique in a planner called *AltAlt* (see next section 7. Specifically, we build the planning graph up to the level that all of the subgoals appear non-mutex, e.g level $lev(S)$, where S is the set of subgoals being considered. The rationale for limiting the level of the graph to $lev(S)$ is that the child states resulted from regressing from the goal state S are likely to have the lev values well

[9]The Lisp source code for fast bi-level planning graph construction is provided by Terry Zimmerman

below $lev(S)$. Therefore, the heuristic estimates based on lev values for these states tend to remain the same (should we expand the graph to level-off). In such cases, expanding the graph further does not necessarily give more information. When a graph is not grown to level-off, the notion of lev has to be redefined as follows:

Definition 5 (Level in a partially constructed graph) *Let l be the index of the last level that the planning graph has been grown to. Given a set S of propositions, denote $lev(S)$ as the index of the first level in the planning graph in which all propositions in S appear and are non-mutexed with one another. If no such level exists, then $lev(S) = l + 1$. Similarly, denote $lev(p)$ as the index of the first level that a proposition p comes into the planning graph.*

In principle, the resulting heuristic may not be as informed as the original heuristic from the full planning graph. We shall see in the section 7 that in many problems the loss of informedness is more than offset by the improved time and space costs of the heuristic.

6.2 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 heuristics for guiding the state search, as mentioned earlier in section 3, 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. `Sel-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 appearing in the action level $lev(S)$ is considered to regress the state S. The intuition behind `sel-exp` strategy is that the actions in level $lev(S)$ comprise the actions that are likely to achieve the subgoals of S in the most direct way from the initial state. While this strategy may in principle result in the incompleteness of the search, because some actions needed for the solution plan that appear much later at levels of index greater than l are not considered, we have not found a single instance in which our strategy fails to find a solution that can be found considering full set of actions. As we shall see in the next section, the `sel-exp` strategy, this strategy has significant effect on the performance of *AltAlt* in some domains such as the Schedule World[1].

We have recently incorporated these ideas into a planner called *AltAlt*, implemented in C, that took part in the AIPS-2000 planning competition [1]. The details on the performances of this planner is described in the section 7.

6.3 Comparing time and space complexity of graph construction vs. dynamic programming style computation

Most of our heuristic functions developed in the previous sections are based upon the level value of some set of propositions, $lev(S)$. This function can be computed using dynamic programming (DP) style computation as well. Specifically, according to the proposition 1, this computation boils down to computing lev function value for all pairs of propositions (assuming that only binary mutexes are computed). The mutex marking and propagation in the planning graph directly correspond to the updating procedure of the lev function. In addition, the convergence of lev values directly corresponds to the lev values extracted from a planning graph that levels off. In fact, in a recent work, Haslum & Geffner [14] have started from the DP formulation to derive a family of heuristics that are closely connected to the set-level heuristic function $h(S) = lev(S)$ developed in section 4. The heuristics used in HSP and HSP-R planners are also computed using bottom-up DP approach.

Given the lev function value, it is simple to see that the heuristics developed in the previous sections can also be computed using the DP style approach. It is straightforward to compute the partitioning heuristic family, as they involve only the level information. The *adjusted-sum* and *combo* heuristics involve computing the level information, as well as the *sum heuristic* value, which were already computed in a DP fashion. The adjusted-sum2 heuristic additionally involves computing $cost_p(S)$, which essentially returns the length of a plan extracted from the planning graph that achieves S from the initial state under the assumption that there are no delete effects of actions. A planning graph that ignores the delete effects of actions can be built cheaply in terms of both time and memory.

The bottom line is that, unlike the Graphplan algorithm, which needs to construct the planning graph structure in order to search for a solution subgraph, we might not need to compute the planning graph in order to compute our heuristics. Thus, one might wonder whether building the planning graph would be more costly both in terms of time and space, compared with using DP style computation. We will show that this is not the case. Specifically, using a bi-level representation of the graph and efficient mutex marking procedures such as those in STAN [31], the time it takes for the building the graph tends to be lower than the time it takes for the updating procedures to converge. Furthermore, the additional amount of memory required to store such a graph is often not very significant.

Indeed, notice that the graph level-by-level expansion can be seen as the updating procedures of lev values. The interesting point is that, only actions that are applicable in the previous level are considered in the updating procedures, and only pairs of propositions whose lev value changes are actually updated. Thus, the expansion of the planning graph can be seen as a selective and directional way of updating the lev values efficiently, improving the convergence time of such values.

Now, in terms of memory. Let P and A be the total number of (grounded) propositions and actions, respectively. Let α, β, and γ be the average amount of memory needed to store a single proposition, a mutex relation, and an action, respectively. Using the DP style computation, only the lev values of all propositions and the pair of propositions – there are $P(P-1)/2$ of those – are stored, taking $(\alpha P + \beta P(P-1)/2)$ amount of memory. In the bi-level representation of planning graph, the set of propositions, pairs of mutexes, and the set of actions are all that need to be stored. These numbers are essentially independent of the number levels the planning graph needs to expand, since the value of any of these entities with respect to each particular level is represented by one bit, not by duplicating as in the original Graphplan's planning graph. Thus, the amount of memory needed to store the bi-level graph structure is $(\alpha P + \beta P(P-1)/2 + \gamma A)$. The additional amount of memory the planning graph actually has to store is the set of all actions, which is γA. It is important to emphasize that this amount is often not very significant, as it is independent of the number of graph levels. Moreover, the set of actions are very helpful in focusing the search (see subsection 6.2).

Therefore, contrary to the initial impressions, the planning graph does not take much more memory than using DP style procedure to come up with the same heuristic functions. Furthermore, that additional amount of memory needed by the planning graph actually stores the action information, which is very useful in improving the relevance of the search. In addition, the graph tends to level off faster than the convergence rate of the lev values using dynamic programming approach. This underscores the fact that the planning graph can serve not only as a powerful *conceptual* model, but also an efficient *implementational* model for computing effective admissible heuristics.

7 AltAlt, a state search planner using planning graph based heuristics

Until now, we concentrated on analyzing the tradeoffs among the various heuristics one can extract from the planning graph. The implementation used was aimed primarily at normalized comparisons among these heuristics. To see how well a planner based on these heuristics will compare with other highly optimized implementations, we built *AltAlt*[10] planner implemented in C programming language. Empirical results show that *AltAlt*[11] can be orders of magnitude faster than *both* STAN and HSP-r, validating the effectiveness of the heuristic being used. Furthermore, it turns out that *AltAlt* is also very competitive with some of the best planners at the planning competition.

AltAlt is implemented on top of two highly optimized existing planners–STAN [31] that is a very effective Graphplan style planner is used to generate planning graphs, and HSP-r [4], a heuristic search planner provides an optimized state search engine. We use one of the most effective heuristics developed earlier to guide the search – the *adjsum2M* heuristic (section 4.3).

The high-level architecture of *AltAlt* is shown in Figure 5. 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 [31] for this purpose as it provides a fast mutex marking routines and 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, described in section 4. 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 [4].

7.1 Evaluating the performance of AltAlt

AltAlt's 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 here 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 (see subsection 7.2).

Our experiments were all done on a Linux system running on a 500 mega hertz 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. We also compare *AltAlt* to FF and HSP2.0, which are two of the best planners at the competition. HSP2.0 is a recent variant of the HSP-R system that opportunistically shifts between regression search (HSP-R) and progression search (HSP). FF is an extremely efficient state search planner using a novel local search style algorithm and many useful pruning techniques. 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,

[10] A Little of this and a Little of That.

[11] An earlier version of *AltAlt* took part in the automated track in the recent AIPS-2000 planning competition[1]. At the time of competition it was still not completely debugged. In the competition, *AltAlt* managed to stay in the middle range among the 12 competitors. The current debugged and optimized version, on the other hand, is able to compete favorably with the top four planners in the competition, including Hoffman's FF[15], Bonet and Geffner's HSP2.0 [5, 4], and Refanidis's GRT [38].

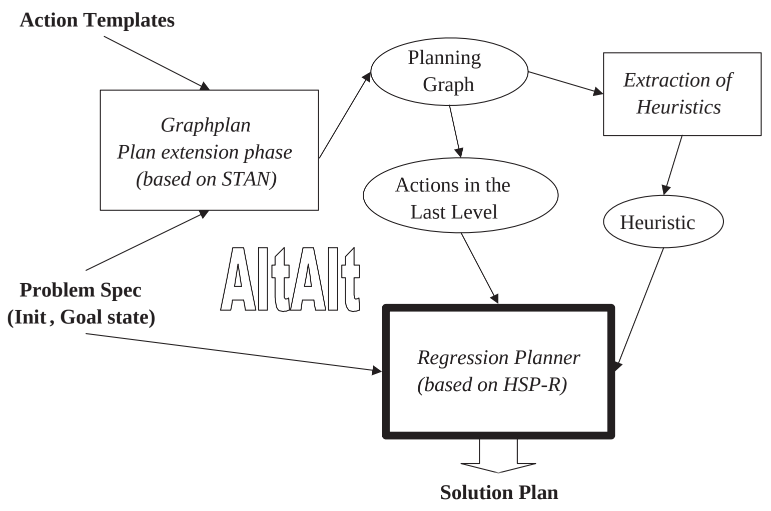


Figure 5: Architecture of *AltAlt*

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 6). For each node representing a state S, only the action instances present in the action level $lev(S)$ of the planning graph are used to expand nodes in the regression search (see Section 6).

Table 6 shows some statistics gathered from head-on comparisons between *AltAlt* , STAN, HSP-R, HSP2.0 and FF 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 and FF. While HSP2.0 predictably outperforms HSP-R, it is still dominated by *AltAlt* , especially in terms of solution quality. FF is clearly shown to be very fast here, but it fails to solve some problems. *AltAlt* gives better solution quality in most problems.

The plots in Figures 6, 7, 8 compare the time performance of STAN, HSP-R, HSP2.0 and *AltAlt* in specific domains. The plot in figure 6 summarizes the problems from blocks world and the plot in figure 7 refers to the problems from logistics domain, and plot in figure 8 to the problems from the scheduling world, three of the standard benchmark domains that have ben used in the recent planning competition [1]. 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. HSP-R is unable to solve any problem in this domain. 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.

7.2 Trading quality for cost in heuristic computation in AltAlt

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 6 showed, deriving heuristics from such partial planning graphs trades cost of the heuristic computation with

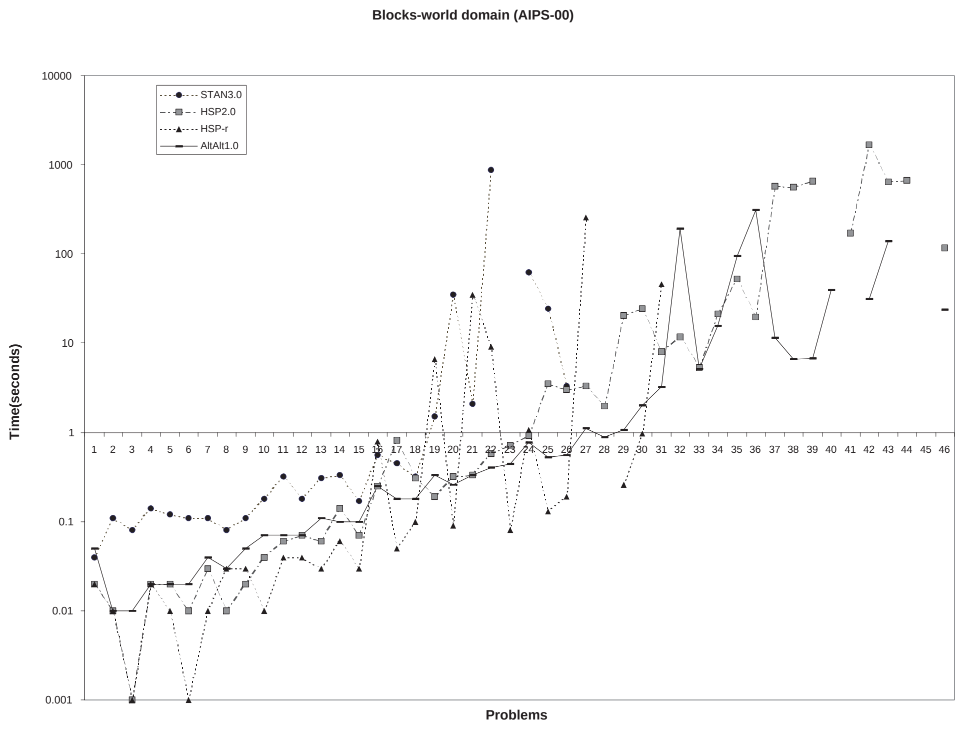


Figure 6: Results in Blocks world domain.

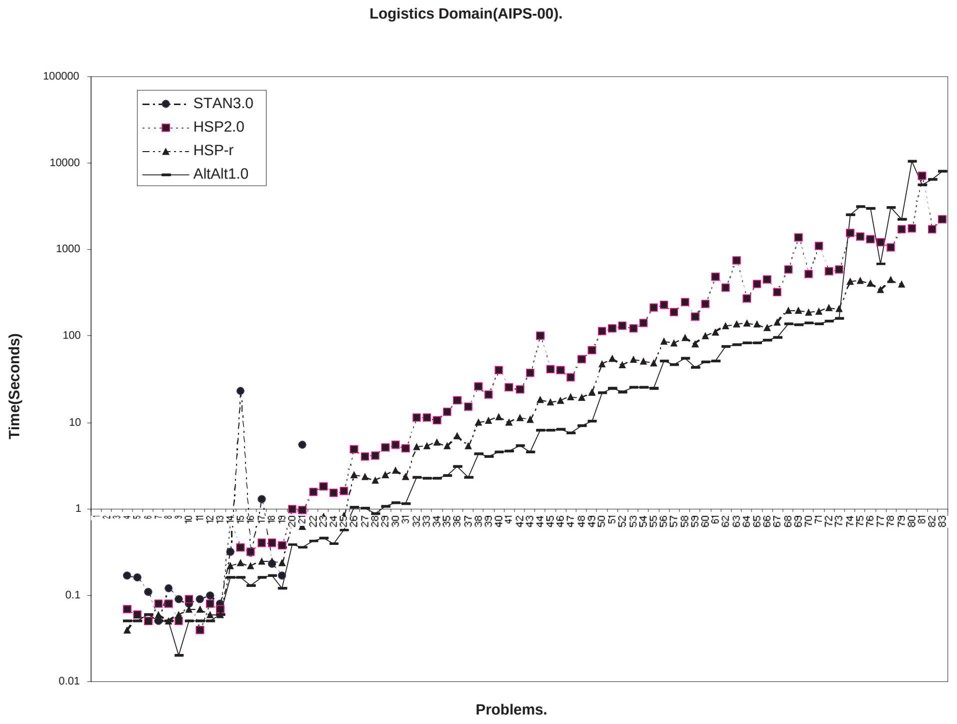


Figure 7: Results in Logistics domain.

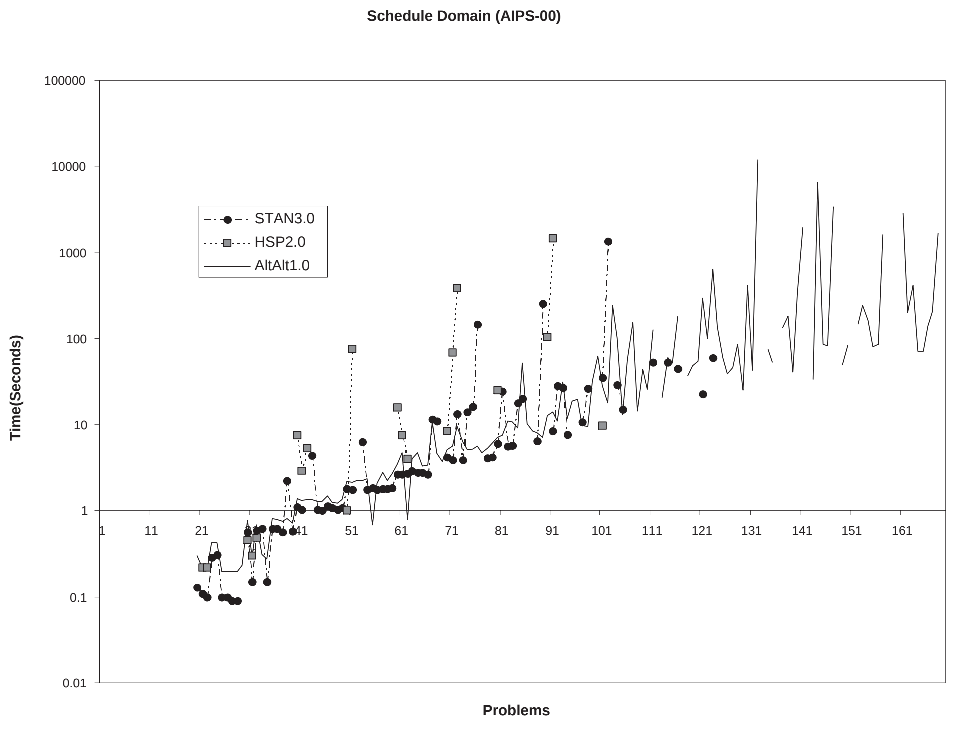
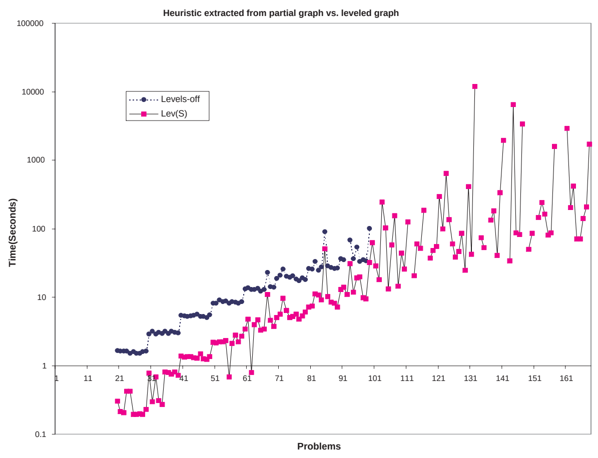
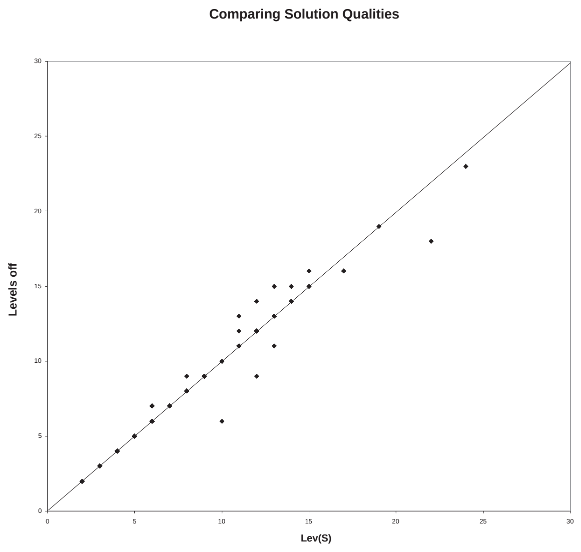


Figure 8: Results in Schedule domain.



(a) Running Time



(b) Solution Quality

Figure 9: Results on trading heuristic quality for cost by extracting heuristics from partial planning graphs.

	STAN3.0		HSP-r		HSP2.0		FF		AltAlt(AdjSum2M)	
Problem	Time	Length	Time	Length	Time	Length	Time	Length	Time	Length
gripper-15	-	-	0.12	45	0.19	57	0.02	45	0.31	45
gripper-20	-	-	0.35	57	0.43	73	0.02	57	0.84	57
gripper-25	-	-	0.60	67	0.79	83	0.03	67	1.57	67
gripper-30	-	-	1.07	77	1.25	93	0.08	77	2.83	77
tower-3	0.04	7	0.01	7	0.01	7	0.02	7	0.04	7
tower-5	0.21	31	5.5	31	0.04	31	0.02	31	0.16	31
tower-7	2.63	127	-	-	0.61	127	0.15	127	1.37	127
tower-9	108.85	511	-	-	14.86	511	1.92	511	48.45	511
8-puzzle1	37.40	31	34.47	45	0.64	59	0.19	47	0.69	31
8-puzzle2	35.92	30	6.07	52	0.55	48	0.15	84	0.74	30
8-puzzle3	0.63	20	164.27	24	0.34	34	0.08	42	0.19	20
8-puzzle4	4.88	24	1.35	26	0.46	42	0.05	44	0.41	24
aips-grid1	1.07	14	-	-	2.19	14	0.06	14	0.88	14
aips-grid2	-	-	-	-	14.06	26	0.29	39	95.98	34
mystery2	0.20	9	84.00	8	10.12	9	0.12	10	3.53	9
mystery3	0.13	4	4.74	4	2.49	4	0.03	4	0.26	4
mystery6	4.99	16	-	-	148.94	16	-	-	62.25	16
mystery9	0.12	8	4.8	8	3.57	8	0.20	8	0.49	8
mprime2	0.567	13	23.32	9	20.90	9	0.14	10	5.79	11
mprime3	1.02	6	8.31	4	5.17	4	0.04	4	1.67	4
mprime4	0.83	11	33.12	8	0.92	10	0.03	10	1.29	11
mprime7	0.418	6	-	-	-	-	-	-	1.32	6
mprime16	5.56	13	-	-	46.58	6	0.10	7	4.74	9
mprime27	1.90	9	-	-	45.71	7	0.41	5	2.67	9

Table 6: Comparing the performance of *AltAlt* with STAN, a state-of-the-art Graphplan system, HSP-R, a state-of-the-art heuristic state search planner, and HSP2.0 and FF, two planning systems that competed at the AIPS-2000 competition.

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

The plots in Figure 9 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 compares 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 161 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. The few points below the diagonal correspond to the problem instances on which the plans generated with the heuristic derived from the partial planning graph were longer than those generated with heuristic derived from the full leveled planning graph. This validates our contention in Section 6 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.

8 Planning Graph-based heuristics for CSP Search, and their application to Graphplan's Backward Search

In the previous sections we have formulated a family of heuristics extracted from the planning graph and showed that these heuristics can be very effective in guiding the **state space search** of the planner. In this section we shall demonstrate that these heuristics can also be applied to improve the Graphplan algorithm's **CSP-style** backward search significantly.

This section is organized as follows. First we shall give a brief review and critique of Graphplan's variable and value ordering in its backward search. Then we shall present several heuristics for variable and value ordering based on the level information on the planning graph. Finally, the effectiveness of these heuristics is evaluated and many of their interesting advantages are discussed.

8.1 Variable and Value ordering in Graphplan's backward search

As briefly reviewed in section 3, the Graphplan algorithm consists of two interleaved phases – a forward phase, where a data structure called "planning-graph" is incrementally extended, and a backward phase where the planning-graph is searched to extract a valid plan.

The search phase on a k level planning-graph involves checking to see if there is a sub-graph of the planning-graph that corresponds to a valid solution to the problem. This involves starting with the propositions corresponding to goals at level k (if all the goals are not present, or if they are present but a pair of them are marked mutually exclusive, the search is abandoned right away, and planning-graph is grown another level). For each of the goal propositions, we then select an action from the level k action list that supports it, such that no two actions selected for supporting two different goals are mutually exclusive (if they are, we backtrack and try to change the selection of actions). At this point, we recursively call the same search process on the $k-1$ level planning-graph, with the preconditions of the actions selected at level k as the goals for the $k-1$ level search. The search succeeds when we reach level 0 (corresponding to the initial state).

Previous work [17, 43, 21] had explicated the connections between this backward search phase of Graphplan algorithm and the constraint satisfaction problems (specifically, the dynamic constraint satisfaction problems, as introduced in [23]). Briefly, the propositions in the planning graph can be seen as CSP variables, while the actions supporting them can be seen as their potential values. The mutex relations specify the constraints. Assigning an action (value) to a proposition (variable) makes variables at lower levels "active" in that they also now need to be assigned actions.

The order in which the backward search considers the (sub)goal propositions for assignment is what we term the **variable ordering** heuristic. The order in which the actions supporting a goal are considered for inclusion in the solution graph is the **value ordering** heuristic. In their original paper, Blum & Furst argue that the goal ordering and value ordering heuristics are not particularly useful for improving Graphplan, mainly because no solution is found in the levels before the solution-bearing level anyway. There are however reasons to pursue goal ordering strategies for Graphplan. First of all, in many problems, the search done at the final level does account for a significant part of the overall search. Thus, it will be useful to pursue variable ordering strategies, even if they improve only the final level search. Secondly, there may be situations where one might have lower bound information about the length of the plan, and using that information, the planning graph search may be started from levels at or beyond the minimum solution bearing level of the planning graph. Indeed, we will show later on that Graphplan's search can be significantly improved using certain types of variable and value ordering heuristics. What is more interesting is that these heuristics are directly drawn from a number of heuristic functions extracted from the planning graph that we had derived earlier for state space search. We shall briefly show the ineffectiveness of the existing heuristics for Graphplan search in the next subsection, motivating the derivation of a family of heuristics extracted from the planning graph.

8.2 The ineffectiveness of existing heuristics for Graphplan's search

The original Graphplan algorithm did not commit to any particular goal or value ordering heuristic. The implementation however does default to a value ordering heuristic that prefers to support a proposition by a noop action, if available. Although the heuristic of preferring noops seems reasonable (in that it avoids inserting new actions into the plan as much as possible), and has mostly gone unquestioned,[12] it turns out that it is hardly infallible. In [20], our experiments with Graphplan implementations show that using noops first heuristic can, in many domains, drastically worsen the performance. Specifically, in most of the problems, considering noops first worsened performance over not having any specific value ordering strategy (and default to the order in which the actions are inserted into the planning graph).

In CSP literature, the standard heuristic for variable ordering involves trying most constrained variables first. A variable is considered most constrained if it has least number of actions supporting it. Although some implementations of Graphplan, such as SGP [43] include this variable ordering heuristic, empirical studies elsewhere have shown that by and large this heuristic leads to at best marginal improvements. In particular, the results reported in [16] show that the most constrained first heuristic leads to about 4x speedup at most.

One reason for the ineffectiveness of most-constrained-first variable ordering is that, the number of actions that can achieve a given goal, i.e the number of values that can be assigned for a given variable, does not adequately capture the "difficulty" of finding an assignment for that variable. Specifically, since a variable with fewer actions supporting it may actually be much harder to handle than another with many actions supporting it, if each of the actions supporting the first one eventually lead to activation of many more and harder to assign new variables, then it is actually the more difficult one to find an assignment. Thus, a strategy such as "most-constrained-variable first" that quickens the process of assigning values to the variables at the current level, may not be effective as it does not concern itself with the question of what types of variables get activated at the lower levels based on the current level assignments.

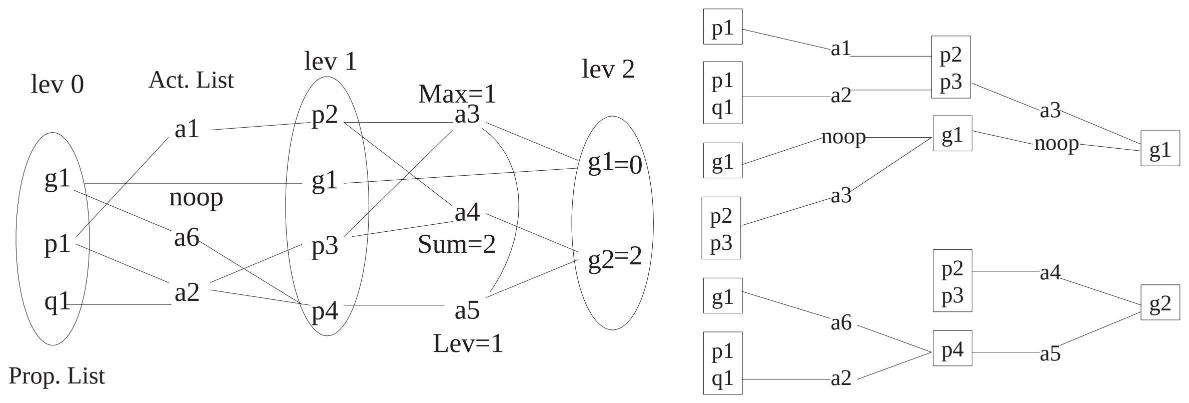
8.3 Extracting heuristics from the planning graph for Graphplan's search

In contrast to standard CSP search, the Graphplan search process does not end as soon as we find an assignment for the current level variables. Instead, the current level assignments activate specific goal propositions at the next lower level and these need to be assigned; this process continues until the search reaches the first level of the planning graph. Therefore, what we need to improve this search is a heuristic that finds an assignment to the current level goals, which is likely to activate fewer and easier to assign variables at the lower levels. Fortunately, the family of state space heuristics developed earlier in section 4 are exactly concerned with this type of information, i.e the difficulty (cost) of achieving a subgoal, or a set of subgoals that are present in the same world state.

To make this connection clearer, let us consider a planning graph in the left of figure 10, and the corresponding CSP-style backward search on the right. Consider the *variable ordering* in a given level, say, level 2 in the planning graph, where we have two variables (goals) g_1 and g_2. Intuitively, since the search is done in the backward direction, we might want to choose the goal that is achieved the last. This is directly related to the difficulty (cost) of achieving a subgoal p from the initial state, which can be captured by the notion of level value of p, namely $lev(p)$, as defined in definition 1.

Considering the *value ordering* given a variable g_2, there are two possible value assignments for g_2, corresponding to two actions a_4 and a_5. The likelihood that such a value assignment would activate easier-

[12] In fact, some of the extensions of Graphplan search, such as Koehler's incremental goal sets idea [22] explicitly depend on Graphplan using noops first heuristic.



(a) Planning Graph (b) (Dynamic) CSP search

Figure 10: A planning graph example and the CSP-style regression search on it. To avoid cluttering, we do not show all the no-ops and mutex relations.

to-assign variables in the next lower levels is directly related to the cost of making the corresponding actions a_4 and a_5 become applicable in a plan executed from the initial state. This cost in turn is directly related to the cost of achieving the set of preconditions of the corresponding actions, namely $Prec(a_4) = \{p_2, p_3\}$ and $Prec(a_5) = \{p4\}$. We can apply the family of heuristic functions developed earlier for measuring the cost of achieving a set of preconditions for a given action.

Now, we are ready to state the variable and value ordering heuristics for Graphplan search:

> *Propositions are ordered for consideration in decreasing value of their levels. Actions supporting a proposition are ordered for consideration in increasing value of their costs. (The cost of actions are defined below)*

These heuristics can be seen as using a “hardest to achieve goal (variable) first/easiest to support action (value) first” idea, where hardness is measured in terms of the level of the propositions.

We use three different heuristic functions developed in section 4 for estimating the cost of actions: *Max, partition-1* and *set-level* heuristic. The reason these heuristics are used is that they are derived directly of the level information extracted from the planning graph and thus very cheap to compute.

Max heuristic: The cost an action is the maximum of the cost (distance) of the individual propositions making up the precondition list of that action, namely, $cost_{Max}(a) = \max_{p \in Prec(a)} lev(p)$. For example, the cost of A_3 supporting G_1 in Figure 10 is 1 because A_3 has two preconditions P_2 and P_3, and both have level 1 (thus maximum is still 1). This heuristic is adapted from the *max heuristic* (see section 2).

Sum heuristic: The cost of an action is the sum of the costs of the individual propositions making up that action’s precondition list, namely, $cost_{Sum}(a) = \sum_{p \in Prec(a)} lev(p)$. For example, the cost of A_4 supporting G_2 in Figure 10 is 2 because A_4 has two preconditions P_2 and P_3, and both have level 1. This heuristic is adapted from the *partition-1 heuristic* (see section 4).

Level heuristic: The cost of an action is the index if the first level at which that set of that action’s preconditions are present and are non-mutex, namely, $cost_{Level}(a) = lev(Prec(a))$. For example, the cost of A_5 supporting G_2 in Figure 10 is 1 because A_5 has one precondition P_4, and it occurs in level 1 of

Problem	Normal GP		Max GP		Lev GP		Sum GP		Speedup		
	Length	Time	Length	Time	Length	Time	Length	Time	Max	Lev	Sum
bw-large-a	12/12	.008	12/12	.005	12/12	.005	12/12	.006	1.6x	1.6x	1.3x
bw-large-b	18/18	.76	18/18	.13	18/18	.13	18/18	.085	5.8x	5.8x	8.9x
BW-large-C	-	>30	28/28	1.15	28/28	1.11	-	>30	>26x	>27x	-
huge-fct	18/18	1.88	18/18	.012	18/18	.011	18/18	.024	156x	171x	78x
bw-prob04	-	>30	8/18	5.96	8/18	8	8/19	7.25	>5x	>3.7x	>4.6x
rocket-ext-a	7/30	1.51	7/27	.89	7/27	.69	7/31	.33	1.70x	2.1x	4.5x
rocket-ext-b	-	>30	7/29	.003	7/29	.006	7/29	.01	10000x	5000x	3000x
att-log-a	-	>30	11/56	10.21	11/56	9.9	11/56	10.66	>3x	>3x	>2.8x
gripper-6	11/17	.076	11/15	.002	11/15	.003	11/17	.002	38x	25x	38x
gripper-8	-	>30	15/21	.30	15/21	.39	15/23	.32	>100x	>80	>93x
ferry41	27/27	.66	27/27	.34	27/27	.33	27/27	.35	1.94x	2x	1.8x
ferry-5	-	>30	33/31	.60	33/31	.61	33/31	.62	>50x	>50x	>48x
tower-5	31/31	.67	31/31	.89	31/31	.89	31/31	.91	.75x	.75x	.73x

Table 7: Effectiveness of level heuristic in solution-bearing planning graphs. The columns titled Level GP, Max GP and Sum GP differ in the way they order actions supporting a proposition. Max GP considers the cost of an action to be the maximum cost of any if its preconditions. Sum GP considers the cost as the sum of the costs of the preconditions and Level GP considers the cost to be the index of the level in the planning graph where the preconditions of the action first occur and are not pair-wise mutex.

the planning graph, for the first time, and it is not mutex with anyone else. This heuristic is adapted from the *set-level* heuristic (see section 4).

It is easy to see that the cost assigned by level heuristic to an action A is just 1 less than the index of the level in the planning graph where A first occurs in the planning graph. Thus, we can think of level heuristic as using the uniform notion of “first level” of an action or proposition to do value and variable ordering.

In general, the Max, Sum and Level heuristics can give widely different costs to an action. For example, consider the following entirely plausible scenario: an action A has preconditions $P_1 \cdots P_{10}$, where all 10 preconditions appear individually at level 3. The first level where they appear without any pair of them being mutually exclusive is at level 20. In this case, it is easy to see that A will get the cost 3 by Max heuristic, 30 by the Sum heuristic and 20 by the Level heuristic. In general, we have: $cost_{Max}(A) \leq cost_{Sum}(A)$ and $cost_{Max}(A) \leq cost_{Level}(A)$, but depending on the problem $cost_{Level}(A)$ can be greater than, equal to or less than $cost_{Sum}(A)$. We have experimented with all three heuristics.

8.4 Evaluation of the effectiveness of level-based heuristics

We have implemented the three level-based heuristics described in the previous section for Graphplan backward search and evaluated its performance as compared to normal Graphplan. Our extensions were based on the version of Graphplan implementation bundled in the Blackbox system [21], which in turn was derived from Blum & Furst’s original implementation. Tables 7 and 8 show the results on some standard benchmark problems. The columns titled “Max GP”, “Lev GP” and “Sum GP” correspond respectively to Graphplan armed with the Max, Level and Sum heuristics for variable and value ordering. Cpu time is shown in minutes. For our Pentium Linux machine 500 MHZ with 256 Megabytes of RAM, Graphplan would normally exhaust the physical memory and start swapping after about 30 minutes of running. Thus, we put a time limit of 30 minutes for most problems (if we increased the time limit, the speedups offered by the level-based heuristics get further magnified).

Problem	Normal GP		Max GP		Lev GP		Sum GP		Speedup		
	Length	Time	Length	Time	Length	Time	Length	Time	Max	Lev	Sum
bw-large-a	12/12	.008	12/12	.006	12/12	.006	12/12	.006	1.33x	1.33x	1.33x
bw-large-b	18/18	.76	18/18	0.21	18/18	0.19	18/18	0.15	3.62x	4x	5x
huge-fct	18/18	1.73	18/18	0.32	18/18	0.32	18/18	0.33	5.41x	5.41x	5.3x
bw-prob04	8/20	30	8/18	6.43	8/18	7.35	8/19	4.61	4.67x	4.08x	6.5x
rocket-ext-a	7/30	1.47	7/26	0.98	7/27	1	7/31	0.62	1.5x	1.47x	2.3x
rocket-ext-b	-	>30	7/28	0.29	7/29	0.29	7/28	0.31	>100x	>100x	>96x
tower-5	31/31	0.63	31/31	0.90	31/31	0.89	31/31	0.88	.70x	.70x	.71x

Table 8: Effectiveness of level-based heuristics for standard Graphplan search (including failing and succeeding levels).

Problem	Normal GP		Max GP		+3 levels		+5 levels		+10 levels	
	Length	Time	Length	Time	Length	Time	Length	Time	Length	Time
bw-large-a	12/12	.008	12/12	.005	12/12	.007	12/12	.008	12/12	.01
bw-large-b	18/18	.76	18/18	.13	18/18	.21	18/18	.21	18/18	.25
BW-large-C	-	>30	28/28	1.15	28/28	4.13	28/28	4.18	28/28	7.4
huge-fct	18/18	1.88	18/18	.012	18/18	0.01	18/18	.02	18/18	.02
bw-prob04	-	> 30	8/18	5.96	-	>30	-	>30	-	>30
rocket-ext-a	7/30	1.51	7/27	.89	8/29	0.006	8/29	0.007	8/29	.009
rocket-ext-b	-	> 30	7/29	.003	9/32	0.01	9/32	.01	9/32	.01
att-log-a	-	>30	11/56	10.21	13/56	8.63	13/56	8.43	13/56	8.58
gripper-6	11/17	.076	11/17	.002	11/17	0.003	11/17	.003	11/15	.004
gripper-8	-	> 30	15/23	.30	15/23	0.38	15/23	0.57	15/23	0.33
ferry41	27/27	.66	27/27	.34	27/27	0.30	27/27	0.43	27/27	.050
ferry-5	-	> 30	33/31	.60	31/31	0.60	31/31	.60	31/31	.61
tower-5	31/31	.67	31/31	.89	31/31	0.91	31/31	.91	31/31	.92

Table 9: Results showing that level-based heuristics are insensitive to the length of the planning graph being searched.

Notice that the heuristics are aimed at improving the search only in the solution bearing levels (since no solution exists in the lower levels anyway). Table 7 compares the effectiveness of standard Graphplan (with noops-first heuristic), and Graphplan with the three level-based heuristics in searching the planning graph containing minimum length solution. As can be seen, the final level search can be improved by 2 to 4 orders of magnitude with the level-based heuristics. Looking at the Speedup columns, we also note that all level-based heuristics have approximately similar performance on our problem set (in terms of cpu time).

Table 8 considers the effectiveness when incrementally searching from failing levels to the first successful level (as the standard Graphplan does). The improvements are more modest when you consider both failing and succeeding levels (see Table 8). This is not surprising since in the failing levels, we have to exhaust the search space, and thus we will do the same amount of search no matter which heuristic we actually use.

The impressive effectiveness of the level-based heuristics for solution bearing planning graphs suggests an alternative (inverted) approach for organizing Graphplan's search–instead of starting from the smaller length planning graphs and interleave search and extension until a solution is found, we may want to start on longer planning graphs and come down. One usual problem is that searching a longer planning graph is both more costly, and is more likely to lead to non-minimal solutions. To see if the level-based heuristics are less sensitive to these problems, we investigated the impact of doing search on planning graphs of length strictly larger than the length of the minimal solution.

Problem	Lev GP						SUM GP					
	+3 levels		+5 levels		+10 levels		+3 levels		+5 levels		+10 levels	
	Length	Time	Length	Time	Length	Time	Length	Time	Length	Time	Length	Time
BW-large-A	12/12	.007	12/12	.008	12/12	0.01	12/12	.007	12/12	.008	12/12	.008
BW-large-B	18/18	0.29	18/18	0.21	18/18	0.24	20/20	0.18	20/20	0.28	20/20	0.18
BW-large-C	28/28	4	28/28	3.9	28/28	4.9	-	>30	-	>30	-	>30
huge-fct	18/18	.014	18/18	.015	18/18	.019	18/18	.014	18/18	.015	18/18	.019
bw-prob04	11/18	18.88	-	>30	-	>30	-	>30	-	>30	-	>30
Rocket-ext-a	9/28	.019	9/28	0.02	9/28	0.02	8/28	.003	8/28	.004	8/28	.006
Rocket-ext-b	9/32	.007	9/32	.006	9/32	0.01	7/28	.011	7/28	.012	7/28	.014
Att-log-a	13/56	8.48	14/56	8.18	13/56	8.45	13/56	8	13/56	8.18	13/56	8.45
Gripper-6	11/15	.003	11/15	.003	11/15	.003	11/15	.004	11/15	.004	11/15	.004
Gripper-8	15/21	0.4	15/21	0.47	15/21	0.4	15/21	0.47	15/21	0.47	15/21	0.4
Ferry41	27/27	0.30	27/27	0.30	27/27	0.34	27/27	0.30	27/27	0.30	27/27	0.34
Ferry-5	31/31	0.60	31/31	0.60	31/31	0.60	31/31	0.59	31/31	0.60	31/31	0.61
Tower-5	31/31	0.89	31/31	0.89	31/31	0.89	31/31	0.87	31/31	0.87	31/31	0.87

Table 10: Performance of Level and Sum heuristics in searching longer planning graphs

Table 9 shows the performance of Graphplan with the Max heuristic, when the search is conducted starting from the level where minimum length solution occurs, as well as 3, 5 and 10 levels *above* this level. Table 10 shows the same experiments with with the Level and Sum heuristics. The results in these tables show that Graphplan with a level-based variable and value ordering heuristic is surprisingly robust with respect to searching on longer planning graphs. We note that the search cost grows very little when searching longer planning graphs. We also note that the quality of the solutions, as measured in number of actions, remains unchanged, even though we are searching longer planning graphs, and there are many non-minimal solutions in these graphs. Even the lengths in terms of number of steps remain practically unchanged–except in the case of the rocket-a and rocket-b problems (where it increases by one and two steps respectively) and logistics problem (where it increases by two steps). (The reason the length of the solution in terms of number of steps is smaller than the length of the planning graph is that in many levels, backward search armed with level-based heuristics winds up selecting noops alone, and such levels are not counted in computing the number of steps in the solution plan.)

This remarkable insensitivity of level-based heuristics to the length planning graph means that we can get by with very rough information (or guess-estimate) about the lower-bound on the length of solution-bearing planning graphs.

A way of explaining this behavior of the level-based heuristics is that even if we start to search from arbitrarily longer planning graph, since the heuristic values of the propositions remain the same, we will search for the same solution in the almost the same route (modulo tie breaking strategy). Thus the only cost incurred from starting at longer graph is at the expansion phase and not at the backward search phase.

It must be noted that the default “noops-first” heuristic used by Graphplan implementations does already provide this type of robustness with respect to search in non-minimal length planning graphs. In particular, the noops-first heuristic is biased to find a solution that winds up choosing noops at all the higher levels–thereby ensuring that the cost of search remains the same at higher length planing graphs. However, as the results in [20] point out, this habitual postponement of goal achievement to earlier levels is an inefficient way of doing search in many problems. Other default heuristics, such as the most-constrained first, or the “consider goals in the default order they are introduced into the proposition list”, worsen significantly when asked to search on longer planning graphs. By exploiting the structure of the planning graph, our level-based heuristics give us the robustness of noops-first heuristic, while at the same time avoiding its inefficiencies.

9 Related work

We have shown that the planning graph provides a rich structure that can be exploited to develop a variety of different effective heuristics for domain-independent planning. The needs for such different domain-independent heuristics was persuasively argued in [26].

In this section, we briefly reflect on some important techniques for deriving effective admissible heuristics in the AI search literature, which provides interesting perspectives on the family of heuristics that we have developed. We specifically show how many of our heuristics can be viewed as adapting from powerful techniques in the search literature into the general setting of planning. This sheds some light into the source of strength of the heuristics developed, as well as some possible avenues for further development of effective and admissible heuristics in planning. We also discuss the interesting relation between deriving effective heuristics to the constraint enforcement of the underlying CSP of the problem in planning.

9.1 Deriving effective admissible heuristics in AI search literature

Extracting effective and admissible heuristics has traditionally been a central problem in the AI search literature [37]. The standard explanation for computing these heuristic functions is to compute the cost of exact solutions to a simplified version of the original problem. For example, in the 8-puzzle problem we solve a relaxed version of the problem by assuming that any tile can be moved to an adjacent square and multiple tiles can occupy in the same square, resulting in the well-known Manhattan heuristic function, which is the sum of all Manhattan distances of each individual tile positions in the initial and goal states. Since each individual tiles can move independently of each another in the relaxed problem, this sum heuristic is admissible. However, this heuristic estimate is still not very accurate when we move to bigger problems such as 15-puzzle, because the interactions among the tiles are not accounted for.

Culberson and Schaeffer[6] introduced the notion of *pattern databases* as a method for accounting for such interactions. For instance, in the 15-puzzle problem, they consider any subset of tiles, such as the seven tiles in the right column and bottom row, which they called the *fringe patterns.* The minimum number moves required to get the fringe tiles from their initial state to the goal state, including any required moves of other (non-patterned) tiles as well, is obviously a lower bound on the distance from the initial state to the goal state of the original problem. Since the number of such patterns are limited, they can be computed in the preprocessing phase and then stored efficiently in the memory. Furthermore, multiple pattern databases can be exploited, and the heuristic function takes on the maximum values in all different patterns stored in a given state. This technique was effectively used for solving problem with large search space such as 15-puzzle [6], Rubik's Cube [28].

The idea of pattern databases can be pushed further by identifying *disjoint patterns*, in the sense that each disjoint pattern involves disjoint subsets of tiles as in the 15-puzzle problem, and only moves involving the tiles in the patterns are counted for each pattern. The summation of the distance values corresponding to each disjoint patterns existed in a given state gives a much more accurate distance estimate of that state than taking the maximum. A trivial example of the disjoint pattern database is the Manhattan distance, wherein each individual disjoint pattern can be seen as a single tile position.

While the summation of disjoint pattern databases provides a very powerful rule for raising the accuracy of the heuristic estimate, the problem with this idea is that it is not easy to identify the disjoint patterns in a given domain. Indeed, the existence of these disjoint patterns requires the actions to affect only subgoals within a given pattern, which is not a general characteristic across different problems. As noted by Korf[30], disjoint databases cannot be used on the Rubik's cube.

In between the two extremes (i.e the max and sum of the pattern databases) lies a technique that attempts to combine the two ideas in a different way. The basic idea is that, as in the example of the 15-puzzle problem, consider a database that contains the number of moves required to correctly position ever pair of tiles. While most of the pairs would return the value which is exactly the Manhattan distance, some special positions return a longer pairwise distance due to the subgoal interactions that are considered explicitly. This increased distance value is added into the Manhattan distance, providing a significant improvement into the Manhattan heuristic which basically assumes the subgoal independence. This is the basic idea behind the *linear-conflict* heuristic function, first proposed by Hansson et al [13]. Korf and Taylor[27] have applied this idea into solving 24-puzzle successfully and along the way they introduced several important ideas for accounting for the subgoal interactions. For instance as in the 24-puzzle, there may be many pairs whose pairwise distances are longer than their respective Manhattan distances. Given a state corresponding to a configuration of tiles positions, the set tiles have to be partitioned into disjoint subsets in a way that maximize the sum of all the pairwise distances. Furthermore, the same idea can be generalized to distances of triples or quadruples of tiles as well.

In the following we draw some parallel on the derivation of heuristics in the AI search literature and domain-independent planning.

9.2 Deriving effective and admissible heuristics in planning: Relations with the search literature

Computing effective and/or admissible heuristics for planning is perhaps even more challenging than in the AI search literature, not because the planning problems present larger search spaces compared to standard problems considered in the search literature, but because the heuristics have to be extracted from a general representation that can accommodate *all* planning domains. In contrast, most accurate admissible heuristics in the search literature often exploit the explicit structure and representation of a particular domain in question that can not be easily applied in other domains. Naturally, one should consider applying the powerful insights for computing effective and admissible heuristics in the AI search literature into the general setting of planning.

Surprisingly, deriving effective heuristics for domain-independent planning started to receive attention only recently. Ghallab and his colleagues first used the graph-based heuristic in IxTeT[11], a plan-space based planner. However the effectiveness of their heuristic was not investigated adequately. Subsequently the idea is independently explored by McDermott[34], and Bonet and Geffner[5]. Their work helped revive the heuristic state search tradition in planning.

All of these early heuristics essentially assume the subgoal independence. For example, Bonet & Geffner's *sum* heuristic can be seen as an adaptation of Manhattan heuristic in the slide puzzle problems. More recently, Refanidis[38] and Hoffman[15] have considered the *positive* interactions while ignoring the negative interactions among subgoals to improve the heuristics in many problem domains. Specifically, Hoffman [15] uses the length of the first relaxed plan found in a relaxed planning graph (without mutex computation) as the heuristic value, which can be seen as a special case of the adjusted-sum2 heuristic $h(S) = cost_p(S) + \Delta(S)$, where $\Delta(S) = 0$, completely ignoring the negative interactions. However, FF uses a novel local search style algorithm, armed with many useful pruning techniques to overcome its heuristic limitations and make it an *extremely* efficient (albeit theoretically incomplete) planner. Refanidis [38] extracts the co-achievement relation among subgoals from the first relaxed plan to account for the positive interactions. In fact, the extraction of the co-achievement relation among subgoals does consider the delete effects of the actions. However, this is not sufficiently considered as accounting for the negative interactions among the subgoals because the *first* relaxed plan corresponding to which the co-achieveness relations are extracted may or may not be the correct plan to be found. These heuristics were reported to provide both significant speedups and improved solution quality. Note that none of these heuristics are admissible.

In our work, we consider both negative and positive interactions among subgoals aggressively. While our heuristic families were independently discovered in the context of domain-independent planning under the general STRIPS representation, it turns out that many of these heuristics are closely related to many powerful techniques in the AI search literature. Our work can be interpreted as having explicitly shown that the planning graph, with its rich structure, can serve as a powerful conceptual model for applying the important techniques in the search literature into computing domain-independent heuristics for planning.

The admissible *set-level* heuristic can actually be seen as being adapted from and inspired by the idea of pattern databases, wherein each pair of subgoals can be seen as a "pattern". The difference here is that the $lev(\{p_1, p_2\})$ is *not* the exact cost (distance) of achieving the pair $\{p_1, p_2\}$, but the graph construction and mutex computation ensures that it provides a lower bound for achieving $\{p_1, p_2\}$ [13]. The set-level heuristic thus involves taking the maximum values among all the "patterns" existing in a given set. Furthermore, the accuracy of the admissible set-level heuristic can be improved by increasing the size of the patterns, corresponding to the computation of the higher-order mutexes. The set-level heuristic is also closely related to the family of heuristics introduced by Haslum and Geffner[14] using dynamic programming style formulation (see also subsection 6.3)..

The improvement of the admissible set-level heuristic by running limited backward search on the Graphplan's planning graph to pick up useful memos can also be seen as an interesting way of actively detecting and evaluating useful patterns, which are to be used in the subsequent state search.

The family of *partition-k* heuristics is related to the disjoint pattern database idea. However, the partition-k heuristics are not guaranteed to be admissible, as it is not clear how to efficiently partition a set into subsets that are strictly independent to each other.

The family of *adjusted-sum* heuristics are related to the works by Hasson et al [13] and Korf et al [27]. These heuristics provide a powerful way of improving the effectiveness as well as quality of the sum heuristic by accounting for both positive and negative subgoal interactions. Specifically, we deal with one type of subgoal interactions by systematically ignoring the other type of interactions. The heuristic functions are composed of two separate components that are responsible for improving the informedness and admissibility of the heuristics. These heuristics are empirically shown to be superior to most existing state space heuristics in the planning literature both in terms of effectiveness and solution quality.

One problem with these heuristics is that the sum heuristic (unlike the Manhattan in the domain-specific slide puzzle problems) is not admissible to begin with. Thus the family of adjusted-sum heuristic is also not guaranteed to be admissible, although they become significantly more informed for accounting for the negative interactions among subgoals. By replacing the sum heuristic function by a value that manages to account for the positive subgoal interactions, we have adjusted-sum2 heuristics that empirically give very close estimate to the goal, and the quality of the solutions they help find was shown to be often optimal or close to optimal in our experiments.

9.3 Relations with consistency enforcement in CSP

Our work also reveals an interesting connection between computing effective heuristics for state space search and the constraintedness of the CSP. In state space search, having *perfect* heuristic function enables the search to find the solution without backtracking. Similarly the solution for a *globally consistent* constraint network can also be found in a backtrack-free manner. This shows the direct connection between the amount of search involved with the informedness of the heuristic estimate in state space search, and the constraintedness of the constraint networks in CSP search.

In planning, a planning problem can be formulated both as a state space search problem as well as

[13]Computing the exact cost of achieving a pair of subgoals may not be any easier than the original problem itself. In fact, may planning problems have only one or two subgoals to begin with.

CSP problem. In particular, the planning graph can be seen as a CSP-based representation of the problem. with the (binary) mutex constraints computation and propagation corresponds directly to a form of directed partial 1- and 2-consistency enforcement. Furthermore, our heuristics are shown to benefit from the computation and propagation of higher order mutexes, corresponding to stronger consistency enforcement of the underlying CSP. We thus exploit the these constraints in the planning graph to improve the informedness of heuristics for the problem's formulation as state space search.

The set of mutex constraints play very important role in improving the informedness of our graph-based heuristics. The *level-specific* mutexes can be used to give finer (longer) distance estimates, while *static* mutexes help prune more invalid and/or unreachable states. Thus, our heuristics can be improved by detecting more mutexes. Indeed, more level-specific mutexes can be discovered through more sophisticated mutex propagation rules [8] , while binary and/or higher order static mutexes can be discovered using a variety of different techniques[12, 39, 10].

While the heuristic derivation was focused mostly on state search planning, in section 8 we have shown that similar insights can be applied to other approach such as Graphplan's backward search, which can be viewed as (dynamic) CSP search. In a separate work, we also showed that the same Graph-based heuristics, coupled with efficient handling of the mutual exclusion constraints among subgoals, can help improve plan-space planners such as the UCPOP[42] remarkably. Indeed, the UCPOP planner, enhanced with effective heuristics was shown to be able to find 100-length plans for planning problems that had previously been untouchable[25].

9.4 Relation with Forward State Search planners

We have presented a family of heuristics for backward state space search. The heuristics are extracted from the planning graph that is built in forward direction from the initial state.

It is the conventional wisdom in planning literature that it is natural to search for the solution in backward direction from goal. Indeed, this also agrees with the goal-directed behavior of intelligent agents that plan. Technically, backward search is favored over forward search because backward search considers only relevant actions to the goal, thus reducing the branching factor. However, most current classical planning systems work with representation such as STRIPS, where only relevant propositions are encoded in each domain in question. Thus the advantage the backward branching factor is no longer obvious. One the other hand, since the goal state are often incomplete, the regression from goal over a sequence of some arbitrary sequence of actions may often result into an inconsistent state that is not easily be detected. Representation language such as STRIPS do not readily reveal any such inconsistency information. For example, in the blocks-world, it is not immediately known upfront that the two propositions *on(a,b)* and *clear(b)* are inconsistent in the first place regardless of the initial state.

This inconsistency can be considered as an extreme form of negative interactions among subgoals, as it says the two (or more) subgoals cannot be achieved together at any point. We have shown that the planning graph can be used to detect this through static (binary) mutexes. We have showed that it is very costly to compute and propagate higher-order static mutexes. Even computing the complete set of binary static mutex is known to be NP-hard. Thus it is computationally prohibitive to prune all inconsistent states encountered in the backward search. This is one fundamental disadvantage of searching in backward direction.

In classical planning, forward search does not encounter this problem at all, because at any point of the progression, the state is always assumed to be complete. This is one of main reason behind the current dominance of forward state search planners over planners that search in backward direction. The question is, can the same insights obtained from extracting heuristics from the planning graph, which are used for backward search, be applied to deriving effective graph-based heuristics for forward search?

McDermott's UNPOP planner [34] actually worked in this direction, where the heuristic function is extracted from a so-called regression graph that is built from the goal state. As mentioned earlier, no subgoal interactions are considered, and thus UNPOP thrashed on problems with such strong subgoal interactions. We conjecture that a similar scheme for mutex constraint propagation can also be applied for a planning graph built from the goal state. And while we no longer have to deal with the inconsistent states, the level-specific (dynamic) mutexes can be exploited to improve the cost estimate of the states in a similar way.

10 Concluding Remarks

In this paper, we showed that the planning graph structure used by Graphplan provides a rich source for deriving heuristics for guiding both state space search and CSP-style search.

In the case of state space search, we described a variety of heuristic families, that use the planning graph in different ways to estimate the cost of a set of propositions. Our empirical studies show that many of our heuristics have attractive tradeoffs in comparison with existing heuristics. In particular, we provided three heuristics– "adjusted-sum2", "adjusted-sum2M" and "combo"–which when used in conjunction with a regression search in the space of states, provide performance that is superior to both Graphplan and existing heuristic state search planners, such as HSP-r. We discussed many ways of improving the cost-quality tradeoffs offered by these heuristics–with the prominent among them being derivation of heuristics from partially grown planning graphs. Our empirical studies demonstrate that this approach often cuts down the cost of heuristic computation significantly, with little discernible loss in the effectiveness of the heuristic. Finally, we showed that *AltAlt*, a hybrid planning system based on our heuristics, is very competitive with the state-of-the-art plan synthesis systems. Specifically, our evaluation of *AltAlt* on the AIPS 2000 planning competition data puts its performance on par with the top tier planners in the competition.

In the case of CSP search, we showed how we can derive highly effective variable and value ordering heuristics, which can be used to drive Graphplan's own backward search. We showed that search with these heuristics has the attractive property of being largely insensitive to the length of the planning graph being searched. This property makes it possible to avoid the exhaustive search in non-solution bearing levels by starting with a planning graph that is longer than the minimal length solution.

Our work also makes several pedagogical contributions. To begin with, we show how the strengths of two competing approaches–heuristic state search and Graphplan planning–can be harnessed together to synthesize a family of planners that are more powerful than either of the base approaches. Secondly, we show that cost estimates of sets of subgoals, that have hither-to been used primarily as the basis for heuristics in the context of state-space search, can also be used in the context of CSP search to give rise to very effective variable and value ordering heuristics. Our discussion of relations between our work and the work on memory-based heuristics in the search community shows how the planning graph can be seen as playing the same role for planning, as pattern databases play for usual state space search problems such as 15-puzzle. In particular, the main attraction of planning graph from the point of view of heuristic derivation is that it provides a systematic and graded way of capturing the subgoal interactions. Finally, our discussion of the effect of mutex constraint propagation on the effectiveness of the heuristics points out the close relation between the degree of consistency of the CSP encoding of a planning problem, and the informedness of the heuristics derived from that planning graph.

Acknowledgements

We thank Minh B. Do, Hector Geffner, Jillian Nottingham, Ioannis Refanidis, David Smith and Biplav Srivastava, for helpful discussions and feedback. Thanks are also due to Terry Zimmerman for providing us with his fast bi-level planning graph expansion code. This research is supported in part by NSF young investigator award (NYI) IRI-9457634, ARPA/Rome Laboratory planning initiative grant F30602-95-C-

0247, Army AASERT grant DAAH04-96-1-0247, AFOSR grant F20602-98-1-0182 and NSF grant IRI-9801676.

References

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

[2] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In *Proc. Principles and Practice of Constraint Programming (CP-95)*, 1995. Published as Lecture Notes in Artificial Intelligence, No. 976. Springer Verlag.

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

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

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

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

[7] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120, 1971.

[8] M. Do, S. Kambhampati and B. Srivastava. Investigating the effect of relevance and reachability constraints on SAT encodings of planning. In *AIPS-2000*, 2000.

[9] D. Frost and R. Dechter. In search of the best constraint satisfactions earch. In *Proc. AAAI-94*, 1994.

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

[11] M. Ghallab and H. Laruelle. Representation and control in IxTeT. In *Proc. AIPS-94*, 1994.

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

[13] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, Vol. 63, No. 3, 1992.

[14] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proc. AIPS-2000*, 2000.

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

[16] S. Kambhampati. 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.

[17] S. Kambhampati, E. Lambrecht, and E. Parker. Understanding and extending graphplan. In *Proc. ECP-97*, 1997.

[18] S. Kambhampati. Challenges in bridging plan synthesis paradigms. In *Proc. IJCAI-97*, 1997.

[19] S. Kambhampati. EBL & DDB for Graphplan. *Proc. IJCAI-99.* 1999.

[20] S. Kambhampati and R.S Nigenda. Distance based goal ordering heuristics for Graphplan. In *AIPS-2000*, 2000.

[21] H. Kautz and B. Selman. Blackbox: Unifying sat-based and graph-based planning. In *Proc. IJCAI-99*, 1999.

[22] J. Koehler. Solving complex planning tasks through extraction of subproblems. In *Proc. 4th AIPS*, 1998.

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

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

[25] X. Nguyen and S. Kambhampati. Heuristic Search Control and Mutual Exclusion Reasoning for Partial Order Planning.

[26] P. Stone, M. Veloso, and J. Blythe. The needs for different domain-independent heuristics. In *AIPS-1994*, 1994.

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

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

[29] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41-78, 1993.

[30] R. Korf. Recent progress in in the design and analysis of admissible heuristic functions (Invited Talk). *Proc. AAAI-2000*, 2000.

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

[32] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proc. AIPS-96*, 1996.

[33] D. McDermott. Aips-98 planning competition results. 1998.

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

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

[36] N. Nilsson. *Principles of Artificial Intelligence*. Tioga, 1980.

[37] J. Pearl. *Heuristics*. Morgan Kaufmann, 1984.

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

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

[40] D. Smith. Private Correspondence, August 1999.

[41] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, San Diego, California, 1993.

[42] D. Weld. An Introduction to Least Commitment Planning. In *AI Magazine*, 1994.

[43] D. Weld, C. Anderson, and D. Smith. Extending graphplan to handle uncertainty & sensing actions. In *Proc. AAAI-98*, 1998.

[44] D. Weld. Recent advances in AI planning. *AI magazine*, 1999.