Extracting Effective and Admissible State Space Heuristics from the Planning Graph

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

Abstract

Graphplan and heuristic state space planners such as HSP-R and UNPOP are currently two of the most effective approaches for solving classical planning problems. These approaches have hither-to been seen as largely orthogonal. In this paper, we show that the planning graph structure that Graphplan builds in polynomial time, provides a rich substrate for deriving more effective heuristics for state space planners. Specifically, we show that the heuristics used by planners such as HSP-R and UNPOP do badly in several domains due to their failure to consider the interactions between subgoals, and that the mutex information in the planning graph captures exactly this interaction information. We develop several families of heuristics, some aimed at search speed and others at optimality of solutions. Our empirical studies show that our heuristics significantly out-perform the existing state space heuristics.

1 Introduction

The last few years have seen a number of attractive and scaleable approaches for solving deterministic planning problems. Prominent among these are "disjunctive" planners, exemplified the Graphplan algorithm of Blum & Furst [1], and heuristic state space planners, exemplified by McDermott's UNPOP [17] and Bonet & Geffner's HSP-R planners [3; 2]. 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 datastructure called "planning graph" [9]. 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 [22].

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 [17; 3].

In this paper, we show that the planning graph datastructure computed in polynomial-time as part of the Graphplan algorithm, provides a general and powerful basis for the derivation of state space heuristics that take subgoal interactions into account. In particular, the so-called "mutex" constraints of the planning graph provide a robust way of estimating the cost of achieving a set of propositions from the initial state. The heuristics derived from the planning graph are then used to guide state space search on the problem, in a way similar to HSP-R [2]. Note that this means we no longer use Graphplan's exponential time CSP-style solution extraction phase.

We will describe several families of heuristics that can be derived from the planning graph structure and demonstrate their significant superiority over the existing heuristic estimators. We will provide results of empirical studies establishing that state space planners using our best heuristics easily outperform both HSP-R and Graphplan planners. Our development focuses both on heuristics that speedup search without guaranteeing admissibility (such as those currently used in HSP-R and UNPOP), and on heuristics that retain admissibility and thus guarantee optimality. In the former case, we will show that our best heuristic estimators are more robust and are able to tackle many problem domains that HSP-R does poorly (or fails), such as the grid, travel, and mystery domains used in the AIPS-98 competition [16].

While our empirical results are themselves compelling, we believe that the more important contribution of our work is the explication of the way in which planning graph can serve as a rich basis for derivation of families of heuristic estimators. It is known in the search literature that admissible and effective heuristics are hard to compute unless the interactions between subgoals are considered aggressively [13]. Our work shows that planning graph and its mutex constraints provide a powerful way to take these interactions into account. Since mutex propagation can be seen as a form of directed partial consistency enforcement on the CSP encoding corresponding to the

*Copyright ©2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

planning graph, our work also firmly ties up the CSP and state search views of planning.

The rest of the paper is organized as follows. Section 2 reviews the HSP-R planner and highlights the limitations of its "sum" heuristic. These limitations are also shared to a large extent by other heuristic state search planners such as UNPOP. Section 3 discusses how the Graphplan's planning graph can be used to measure the subgoal interactions. Section 4 is the heart of the paper. It develops several families of heuristics, some aimed at search speed and some at solution optimality. Each of these heuristic families are empirically evaluated in comparison to HSP-R heuristic and their relative tradeoffs are explicated. Section 5 discusses the related work, and section 6 summarizes the contributions of our work.

2 Limitation of the HSP-R's sum heuristic

HSP-R [2] is currently one of the fastest heuristic state search planners. It casts planning as search through the *regression space* of world states [19]. In regression state space search, the states can be thought of as sets of *subgoals*. 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) $h(S) := \sum_{p \in S} h(p)$

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:

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

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 [2] 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) $h(S) := \max_{p \in S} h(p)$

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* (aka 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. 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:

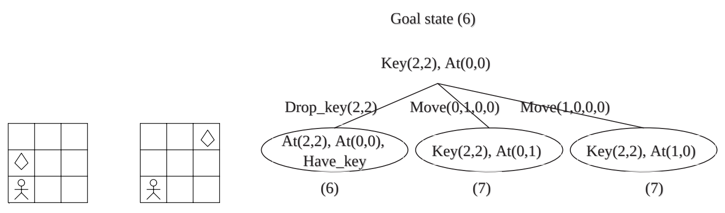
Heuristic 3 (Sum Mutex heuristic)
$h(S) := \infty \ if \ \exists_{p,q \in S} \ s.t. \ mutex(p,q) \ else \ \sum_{p \in S} h(p)$

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[2]. 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 [2], 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 [17] 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: When key(2,2) is first achieved, at(0,0) is no longer true. 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



(a) Grid Problem (b) Regression Search

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

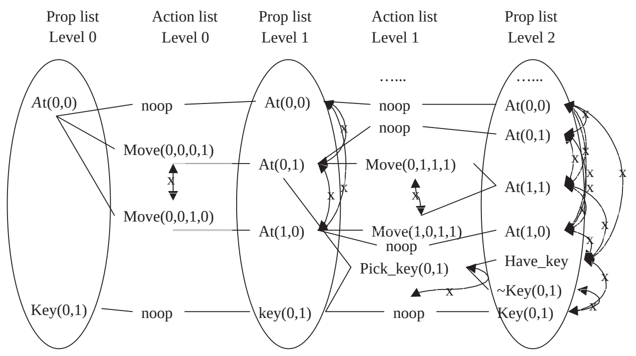


Figure 2: Planning Graph for the 3x3 grid problem

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 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 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, and grid, blocks world, eight puzzle domains[16].

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 weath of mutex constraints, contains much of such information, and can be used to compute more effective heuristics.

Graphplan algorithm [1] 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 statically interfering or if one of the propositions (preconditions) supporting the first action is mutually exclusive with one of the propositions supporting the second action. Figure 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 [11; 22], with the mutex propagation corresponding to a form of directed partial 1- and 2-consistency enforcement [11]. 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** [9]. 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 heuristics from planning graph

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 [16]. 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. 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.

Subsection 4.1 is concerned with effective heuristics without consideration of the solution optimality. 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]. Subsection 4.2 is concerned with improving admissible heuristics for finding optimal solutions. To make the comparisons meaningful, all the planners are implemented in Allegro commonlisp, and share most of the critical data structures. The empirical studies are conducted on a 500 MHz Pentium-III with 512 meg RAM, running Linux. All the times reported include both heuristic computation time and search time, unless specified otherwise.

4.1 Extracting effective heuristics

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.

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. 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) $h(S) := lev(S)$

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-

[1] See [14] for the role of w in BFS.

Problem	sum-mutex	set-lev	adj-sum
bw-large-c	$>$500000	$>$500000	8224
rocket-ext-a	769	$>$500000	658
att-log-a	2978	$>$500000	2224
gripper	930	$>$500000	840
8puzzle-2	1399	51561	1540
8puzzle-3	2899	1047	1384
travel-1	4122	25	40
grid3	$>$200000	49	1151
grid4	$>$200000	44	1148
aips-grid1	$>$200000	108	835
mprime-1	$>$500000	125	96

Table 1: Number of nodes (states) generated by different heuristics, excluding those pruned by mutex computation

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 derived by generalizing the set-level heuristic. The first family, called "partition-k" heuristics, attempt to improve the estimate of 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 by considering the interactions among subgoals. These are described in the next two subsections.

4.1.1 Partition-k heuristics

To avoid underestimating 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). Ideally, we want a partitioning such that elements within each subset S_i may be interacting with each other, but the subsets are independent of each other. Thus we have the following heuristic:

Heuristic 5 (Partition-k heuristic) $h(S) := \sum_{S_i} lev(S_i)$, *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 $k = 1$, 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 2).

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 partition scheme as follows: The basic idea is, in order to avoid underestimating, we put propositions of greatest levels into different partitions. Given a set $S = \{p_1, p_2, ..., p_n\}$. Suppose $lev(p_1) \leq lev(p_2) \leq ... \leq lev(p_n)$. We partition

$$S = \{p_1, p_n\} \cup \{p_2, p_{n-1}\} \cup ... \cup \{p_{[(n-1)/2]}, p_{[(n+1)/2)]}\}.$$

As Table 2 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*.

It would be interesting to have a fuller account of behavior of the family of *partition-k heuristics* with respect to different problem domains. Another related idea is to consider "adaptive partition" heuristics that do not insist on equal sized partitions. For example, p and q are put in the same partition if and only if they are mutexes in the planning graph. We intend to pursue these ideas in future work.

4.1.2 Adjusted Sum Heuristics

We now consider improving the sum heuristic by considering both negative and positive interactions among propositions. 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.

We are now interested in estimating the cost $cost(S)$ for achieving a set $S = \{p_1, p_2, ..., p_n\}$. As before, 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 $S - p_1$. Now, we assume that there are still 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:

$$cost(S) := cost(S - p_1) + cost(p_1) + (lev(S) - lev(S - p_1)) \qquad (2)$$

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

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

[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-b	18/ 379.25	18/ 132.50	18/ 10735.48	-	18/ 79.18	22/ 65.62	22/ 63.57	18/ 87.11
bw-large-c	-	-	-	-	-	30/ 724.63	30/ 444.79	28/ 738.00
bw-large-d	-	-	-	-	-	-	-	36/ 2350.71
rocket-ext-a	-	36/ 40.08	-	32/ 4.04	32/ 10.24	40/ 6.10	34/ 4.72	40/ 43.63
rocket-ext-b	-	34/ 39.61	-	32/ 4.93	32/ 10.73	36/ 14.13	32/ 7.38	36/ 554.78
att-log-a	-	69/ 42.16	-	65/ 10.13	-	63/ 16.97	65/ 11.96	56/36.71
att-log-b	-	67/ 56.08	-	69/ 20.05	-	67/ 32.73	67/ 19.04	61/53.28
gripper-20	-	59/ 90.68	-	59/ 39.17	-	59/ 20.54	59/ 20.92	59/38.18
8-puzzle1	31/ 2444.22	33/ 196.73	31/ 4658.87	35/ 80.05	47/ 172.87	39/ 78.36	39/ 119.54	31/ 143.559
8-puzzle2	30/ 1545.66	42/224.15	30/ 2411.21	38/ 96.50	38/ 105.40	42/ 103.70	48/ 50.45	30/ 348.27
8-puzzle3	20/ 50.56	20/ 202.54	20/ 68.32	20/ 45.50	20/ 54.10	24/ 77.39	20/ 63.23	20/ 62.56
travel-1	9/ 0.32	9/ 5.24	9/ 0.48	9/ 0.53	9/ 0.62	9/ 0.42	9/ 0.44	9/ 0.53
grid3	16/ 3.74	-	16/ 14.09	16/ 55.40	16/ 46.79	18/ 21.45	19/ 18.82	16/ 15.12
grid4	18/ 21.30	-	18/ 32.26	18/ 86.17	18/ 126.94	18/ 37.01	18/ 37.12	18/ 30.47
aips-grid1	14/ 311.97	-	14/ 659.81	14/ 870.02	14/ 1010.80	14/ 679.36	14/ 640.47	14/ 739.43
mprime-1	4/ 17.48	-	4/ 743.66	4/ 78.730	4/ 622.67	4/ 76.98	4/ 79.55	4/ 722.55

Table 2: Number of actions/ Total CPU Time in seconds. The dash (-) indicates that no solution was found in 3 hours or 250MB.

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

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. Therefore, we have

$$h_{adjsum}(S) \approx h_{sum}(S) + h_{lev}(S) - h_{max}(S)$$

It is simple to see that when there is strictly no negative interactions among propositions, $h_{lev}(S) = h_{max}(S)$. Thus, in the formula for $h_{adjsum}(S)$, $h_{sum}(S)$ is the estimated cost of achieving S under the *independence* assumption, while $h_{lev}(S) - h_{max}(S)$ accounts for the additional cost incurred by the *negative* interactions.

Note that the solutions solved by adjusted sum 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 other two terms).

Since there are no 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 are more than one, 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 recurrent relation (assuming unit cost for the selected action a_S)

$$cost_p(S) := 1 + cost_p(S + Prec(a_S) - Add(a_S)) \quad (3)$$

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 $lev(S_0) = 0$, whose cost $cost_p(S_0)$ 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 (and is inspired by) the "relaxed plan" heuristic recently proposed by Hoffman[8].

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)
$h(S) := cost_p(S) + (lev(S) - \max_{p_i \in S} lev(p_i))$*, where* $cost_p(S)$ *is computed using equation (3).*

Table 2 shows that adjusted-sum2 heuristic can solve all types of problem considered. The heuristic is only slightly worse compared with the adjusted-sum 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 complement set of domains (see table 2). 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.*

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 3: 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**

Surprisingly, as shown in table 2 the Combo heuristic is even slightly faster than adjusted-sum heuristic across all type of problems while the solution quality remains comparable.

4.2 Finding optimal plans with admissible heuristics

We now focus on admissible heuristics that can be used to produce optimal plans. Traditionally, efficient generation of optimal plans has received little attention in the planning community. In [9] Kambhampati *et. al.* point out that Graphplan algorithm is guaranteed to find optimal plans when the planning graph serial. In contrast, none of the known efficient state space planners [17; 3; 2; 20] can guarantee optimal solutions.

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 [3], Bonet et al. introduced the *max heuristic* that is admissible. In the previous section, we introduced the *set-level* heuristic that is admissible and showed that it is significantly better 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 3, and clearly establish that set-level heuristic is significantly more effective than max heuristic. Grid, travel, mprime are domains where the set-level heuristic gives very close estimates (see table 2). Optimal search is less effective in domains such as the 8-puzzle and blocks world problem. Domains such as logistics, gripper remain intractable under reasonable limits in time and memory.

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-level mutexes can improve the effectiveness of the set-level heuristic.

Propagating all higher level mutexes is likely to be an infeasible idea [1; 9] (as it essentially amounts to full consistency enforcement of the underlying CSP). A seemingly zanier idea is to use a limited run of Graphplan's own backward search, armed with EBL [11], to detect higher level mutexes in the form of "memos". We have done this 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 level.

Problem	Normal PG	Bi-level PG	Speedup
bw-large-b	22/ 63.57	28/ 20.05	3x
bw-large-c	30/ 444.79	38/ 114.88	4x
bw-large-d	-	44/11442.14	100x
rocket-ext-a	34/ 4.72	34/ 1.26	4x
rocket-ext-b	32/ 7.38	34/ 1.65	4x
att-log-a	65/11.96	64/ 2.27	5x
att-log-b	67/ 11.09	70/ 3.58	3x
gripper-20	59/ 20.92	59/ 7.26	3x
8puzzle-1	39/ 119.54	39/ 20.20	6x
8puzzle-2	48/ 50.45	48/ 7.42	7x
8puzzle-3	20/ 63.23	20/ 10.95	6x
travel-1	9/ 0.44	11/ 0.12	4x
grid-3	19/ 18.82	17/ 3.04	6x
grid-4	18/ 37.12	18/ 14.15	3x
aips-grid-1	14/ 640.47	14/ 163.01	4x
mprime-1	4/ 79.55	4/ 67.75	1x

Table 4: Total CPU time improvement from efficient heuristic computation for **Combo** heuristic

Table 3 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 3). Further analysis is needed to explain these results.

5 Discussion on related work

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 [15], as well as (ir)relevance detection techniques such as RIFO [18] that ignore irrelevant 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. Preliminary 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 (See table 4).

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 [4] , while binary and/or higher order static mutexes can be discovered using a variety of different techniques[6; 21; 5].

Several researchers [8; 20] have considered the *positive* interactions while ignoring the negative interactions among subgoals to improve the heuristics in many problem domains. Hoffman [8] uses the length of the first relaxed plan found in a relaxed planning graph (without mutex computation) as the heuristic value. Refanidis [20] essentially extracts the co-achieveness relation among subgoals from the first relaxed plan to account for the positive interactions. These heuristics were reported to provide both significant speedups and improved solution quality.

Concomitant with our work, Haslum & Geffner [7] considered computing admissible state space heuristic based on dynamic programming approach. Interestingly, their most effective *max-pair* heuristic is closely related to our admissible set-level heuristic. Specifically, the heuristic value updating rule in max-pair heuristic has an effect similar to that of the mutex propagation procedure in the planning graph.

Finally, we concentrated on using the heuristics extracted from the planning graph to drive a state search procedure. In contrast, [12] considers the possibility of using such heuristics to drive Graphplan's own backward search. Their results show that some of the same ideas can be used to derive effective variable and value ordering strategies for Graphplan.

6 Conclusion

In this paper, we showed that the planning graph structure used by Graphplan provides a rich source of effective as well as admissible heuristics. 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-sum", "adjusted-sum2" and "combo" that are clearly superior to the sum mutex heuristic used by HSP-R across a large range of problems, including those that have hither-to been intractable for HSP-R. State search planners using these heuristics out-perform both HSP-R and Graphplan. We are also one of the first to focus on finding effective *and* admissible heuristics for state search planners. We have shown that the set-level heuristic working on the normal planning graph, or a planning graph adorned with a limited number of higher level mutexes is able to provide quite reasonable speedups while guaranteeing admissibility.

Our approach provides an interesting way of incorporating the strength of two different planning regimes (disjunctive vs. conjunctive search) [10] and views (planning as CSP vs. planning as state search) that have hither-to been considered orthogonal. We use the efficient directed consistency enforcement provided by the Graphplan's planning graph construction to develop heuristics capable of accounting for subgoal interactions. We then use the heuristics to guide a state search engine. In contrast to Graphplan, our approach is able to avoid the costly CSP-style searches in the non-solution bearing levels of the planning graph. In contrast to HSP-R and UNPOP, our approach is able to provide much more informed heuristics that take subgoal interactions into account in a systematic fashion.

Acknowledgements. We thank Minh B. Do, Biplav Srivastava, Romeo S. Nigenda, Hector Geffner and Ioannis Refanidis for helpful discussions and feedbacks. Thanks are also due to Terry Zimmerman for providing with us 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] A. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*. 90(1-2). 1997.

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

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

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

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

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

[7] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. To appear in *AIPS-2000*, 2000.

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

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

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

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

[12] S. Kambhampati and R.S Nigenda. Distance based goal ordering heuristics for Graphplan. To appear in *AIPS-2000*, 2000.

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

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

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

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

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

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

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

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

[21] J. Rintanen. An iterative algorithm for synthesizing invariants. To appear in *AAAI-2000*, 2000.

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