ELSEVIER

Artificial Intelligence 135 (2002) 73–123

Artificial Intelligence

www.elsevier.com/locate/artint

Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search ☆

XuanLong Nguyen[1], Subbarao Kambhampati*, Romeo S. Nigenda

Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-5406, USA

Received 18 September 2000; received in revised form 7 September 2001

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 polynomial-time planning graph structure that the Graphplan builds 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 comparisons show that our heuristics handily outperform 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 *AltAlt*, a planner literally cobbled together from the implementations of Graphplan and state search style planners using our theory. We evaluate *AltAlt* 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. © 2001 Published by Elsevier Science B.V.

☆ Preliminary versions of parts of this work have been presented at AAAI-2000 [38], AIPS-2000 [25] and KBCS-2000 [40].

* Corresponding author.

E-mail addresses: xuanlong@cs.berkeley.edu (X. Nguyen), rao@asu.edu (S. Kambhampati), rsanchez@asu.edu (R.S. Nigenda).

[1] Current address: Computer Science Division, University of California, Berkeley, CA 94720, USA.

0004-3702/01/$ – see front matter © 2001 Published by Elsevier Science B.V.
PII: S0004-3702(01)00158-8

Keywords: Planning; Heuristics; Planning graph; CSP; Graphplan

1. Introduction

In the last few years, the planning community has developed a number of attractive and scalable approaches for solving deterministic planning problems. Prominent among these are “disjunctive” planners, exemplified the Graphplan algorithm of Blum and Furst [2], and heuristic state space planners, exemplified by McDermott’s UNPOP [33] and Bonet and Geffner’s HSP-r planners [3,4]. The Graphplan algorithm can be seen as solving the planning problem using CSP techniques. A compact k-length CSP encoding of the planning problem is generated using a polynomial-time data structure called “planning graph”, which is then searched for a solution [20]. On the other hand, UNPOP, HSP, HSP-r are simple state space planners which explicitly search in the space of world states. Their superior performance comes from the heuristic estimators they use to evaluate the goodness of children states. As such, it is not surprising that heuristic state search planners and Graphplan based planners have generally been seen as competing approaches [46].

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 at dealing with problems where there are a lot of interactions between actions and subgoals. Also, Graphplan guarantees, theoretically, to find an optimal solution if one 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 [4,33].

In this paper, we provide a way of successfully combining the advantages of the Graphplan and heuristic state search approaches. Our main insight is that the planning graph structure can be used 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*) 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 propagation of binary

(and higher order) mutual exclusion relations makes 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 which 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 [3]) 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 significantly reduce 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*. *AltAlt* is literally cobbled together from the implementations of Graphplan and state search style planners using our theory. We 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, similar to 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 [19]. 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. This is accomplished 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) [28] that are currently popular in the AI 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 con-

nections 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 subgoal interactions. Section 4 develops several families of effective heuristics, which aim at search speed without insisting on admissibility. Section 5 focuses on generating the admissible heuristic for optimal planning. Several implementational issues of computing the heuristic functions are 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 heuristic guided state space search

Planning can be naturally seen as a search through the space of world states [39]. Unless stated otherwise, in this paper, we shall assume the simple STRIPS [12] model of classical planning. [2] In this model, 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, and a set of deterministic actions Ω. Each action $a \in \Omega$ has a precondition list, add list and delete list, denoted by $Prec(a)$, $Add(a)$, $Del(a)$, respectively. The planning problem is concerned with finding a plan, e.g., a sequence of actions in Ω, that when executed from the initial state S_0 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 $Add(a) \cap S \neq \emptyset$ and $Del(a) \cap S = \emptyset$. 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.

[2] Our theory and techniques work with little change for simple extensions to STRIPS model, such as actions with negated preconditions, and a single effect list consisting of positive and negative literals (instead of delete and add lists).

The efficiency of state search planners and the quality of the solutions that they return depend critically on the informedness and admissibility of these heuristic estimators, respectively. The difficulty of achieving the desired level of informedness and admissibility of the heuristic estimates is due to the fact that subgoals interact in complex ways. There are two kinds of subgoal interaction: 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, making 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, making 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 examining the weakness of heuristics such as those used in HSP-r [3], which ignore these subgoal interactions. Similar arguments can also be generalized to other state space planners such as HSP and UNPOP.

2.1. Importance of accounting for subgoal interactions: Case study of HSP-r

HSP-r planner casts 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 the 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 function h 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). $h_{sum}(S) \leftarrow \sum_{p \in S} h(p)$.

The heuristic cost $h(p)$ of an individual proposition p is computed using an iterative procedure that is run to fixpoint as follows. Initially, each proposition p is assigned a cost 0 if it is in the initial state I, and ∞ otherwise. For each action a that adds some proposition p, $h(p)$ is updated as:

$$h(p) \leftarrow \min\bigl\{h(p), 1 + h\bigl(Prec(a)\bigr)\bigr\}. \tag{1}$$

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 (because achieving some subgoal may also help achieving other subgoals). *Sum* heuristic is also less informed (significantly underestimating) when there are negative interactions between subgoals (because achieving a subgoal deletes other subgoals). Bonet and Geffner [3] 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). $h_{max}(S) \leftarrow \max_{p\in S} h(p)$.

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 [2]. 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 relations can be seen as evidence of 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).

$$\begin{aligned} h(S) \leftarrow\ & \infty && \textit{if } \exists p, q \in S \textit{ s.t. mutex}(p,q) \\ & \sum_{p\in S} h(p) && \textit{otherwise}. \end{aligned}$$

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.

We now provide a brief summary of the procedure for computing mutexes used in HSP-r [3]. 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 computed as the union of sets M_A and M_B, defined as follows: M_A is the set 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 M_B is the set of all pairs $\langle r, q\rangle$, such that for some $\langle p, q\rangle \in M_A$ and for some action a, $r \in Prec(a)$ and $p \in Add(a)$. This computation precludes from consideration potential mutexes $\langle r, s\rangle$, where r and s are not in the add, precondition or delete lists of any single action. As we shall see below, this turns out to be an important limitation in several domains.

The *sum mutex heuristic* used by HSP-r, while shown to be powerful in domains where the subgoals are relatively independent such as the logistics and gripper domains [3], thrashes badly on problems where there is a rich interaction between actions and subgoal sequencing. Specifically, when a subgoal can be achieved early but gets deleted much later when other subgoals are achieved, the *sum* heuristic is unable to account for this interaction. To illustrate this, consider a simple problem from the grid domain [33] shown in Fig. 1. Given a 3×3 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 $key(2,2)$. 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 re-achieved. There are three possible actions: the robot moves from one square to an adjacent square, the robot picks up a key if there is key in the square the robot currently resides, and the robot drops the key at the current square. One obvious solution is as follows: The robot goes from

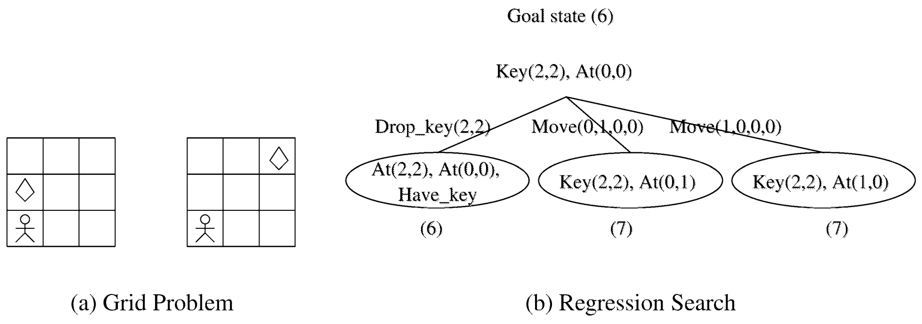

Fig. 1. A simple grid problem and the first level of regression search on it.

$(0,0)$ 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 re-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 (250 MB) on this problem.

It is easy to see how HSP-r goes wrong. First of all, according to the mutex computation procedure that HSP-r uses (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 Fig. 1(b)). 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 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 (both of which are 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 dropping the key at $(2,2)$). 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 than 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 [32].

3. Exploiting the structure of Graphplan's planning graph

In the previous section, we showed the types of problems where ignoring the (negative) interaction between subgoals in the heuristic often leads the search into wrong directions.

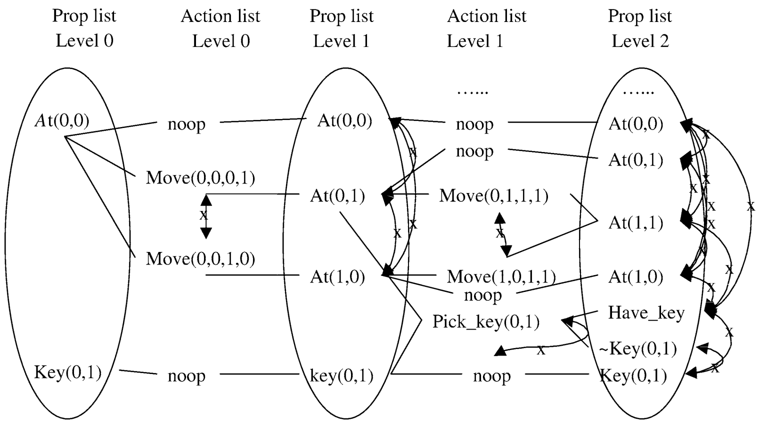

Fig. 2. Planning graph for the 3 × 3 grid problem.

On the other hand, Graphplan's planning graph contains such information in the form of mutex constraints, and can be used to compute more effective heuristics.

The Graphplan algorithm [2] works by converting the planning problem specification into a planning graph. Fig. 2 shows part of the planning graph constructed for the 3×3 grid problem shown in Fig. 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 *and* not mutex in the kth 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 kth 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. The planning graph maintains dependency links between the actions at the level $k+1$, their preconditions in the level k proposition list and their effects in the 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 pairs of actions that have static interference[3] labeled *mutex*. Mutexes are then propagated from this level forward by using two simple propagation rules:

(1) 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.
(2) Two actions at level $k+1$ are mutex if they have static interference or if one of the propositions (preconditions) supporting the first action is mutually exclusive with one of the propositions supporting the second action.

[3] Two actions have a static interference if the union of their preconditions and effects is inconsistent. A set is considered inconsistent if it contains a literal p as well as its negation.

Fig. 2 shows a part of the planning graph for the robot problem specified in Fig. 1. The curved lines with x-marks denote the mutex relations. The planning graph can be seen as a CSP encoding [22,46], with the mutex propagation corresponding to a form of directed partial 1- and 2-consistency enforcement [22]. *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 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* [20]. 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.

Definition 1. A mutex relation is called *static* (or "persistent") if it remains a mutex at the level where the planning graph levels off. A mutex relation is called *dynamic* (or level specific) if it is not static.

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 index of the smallest proposition level in the planning graph in which both propositions appear without a mutex relation.
(2) Any pair of propositions having a static mutex relation between them can never be achieved starting from the 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 leveled planning graph can be used to guide state search planners. The first observation shows that the level information in the planning graph can be used to estimate the cost of achieving a set of propositions. Furthermore, the set of *dynamic* propositional mutexes help give a finer distance estimate. The second observation allows us to prove certain world states that are unreachable from the initial state. The third observation shows a way of extracting a finer (smaller) set of applicable actions to be considered by the regression search.

4. Extracting effective state space heuristics from planning graph

We shall describe how a family of state space heuristics can be extracted from the planning graph. This section is concerned with improving the effectiveness of the heuristics

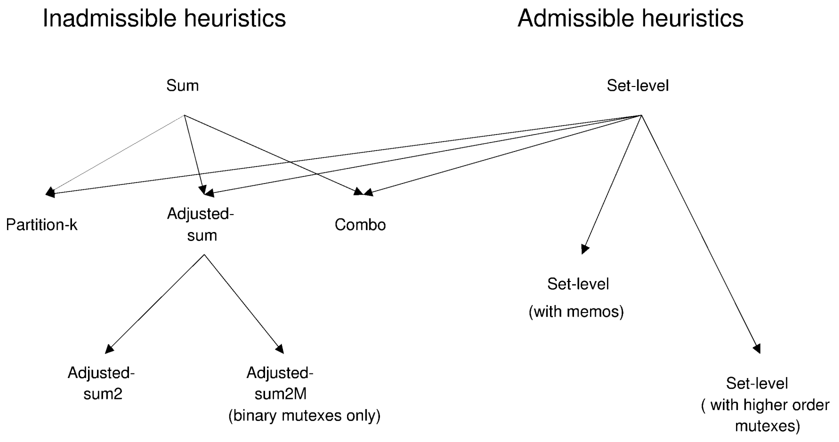

Fig. 3. A road map of heuristics extracted from planning graph.

and the solution quality without insisting on strict admissibility of the heuristic functions. Section 5 is concerned with improving admissible heuristics for finding optimal solutions.

Fig. 3 gives a high level road map of the heuristics to be explored. We use a phased exposition where in we present and evaluate one family of heuristics, and use the limitations discovered by the evaluation to motivate the exploration of the next family of heuristics. To this end, we start by briefly describing how these heuristics are evaluated. We test the heuristics on a variety of planning domains. These include several well-known benchmark domains such as the blocks world, rocket, logistics, 8-puzzle, gripper, mystery, grid and travel. Some of these were used in the AIPS-98 [32] and AIPS-00 competition [1]. 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 blocks world and 8-puzzle domains also have very rich interactions between actions and subgoal sequencing.

All the heuristics extracted from the planning graph as well as 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 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 $w = 5$. [4] 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 MB RAM, running Linux. All the times reported include both heuristic computation time and search time, unless specified otherwise.

[4] See [3,26] for a discussion on the effect of different relative weighting of g and h components using w in best first search.

4.1. Set-level heuristic

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).

Definition 2 (*Level*). Given a set S of propositions, $lev(S)$ denotes the index of the first level in the *leveled serial* planning graph in which all propositions in S appear with no subset of S marked mutex. If no such level exists, then $lev(S) = \infty$. We also write $lev(p)$ for $lev(\{p\})$.

Traditionally, only binary mutexes are computed in the Graphplan algorithm's planning graph. However, mutexes up to order (size) k $(k = 3, 4, \ldots)$ can also be computed. Given a fixed k, the planning graph can be built in polynomial time in terms of the problem size. Therefore $lev(S)$ can be computed in polynomial time. Nevertheless, it should be noted that the cost of building the planning graph increases exponentially in terms of k.

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

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

If mutexes up to order k are present in the planning graph, and $|S| \geqslant k$, then:

$$lev(S) = \max_{p_1, p_2, \ldots, p_k \in S} lev\bigl(\{p_1, p_2, \ldots, p_k\}\bigr).$$

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

It takes only a small step from the observations made in Section 3 to arrive at our first heuristic derived from the planning graph:

Heuristic 4 (Set-level heuristic). $h_{lev}(S) \leftarrow lev(S)$.

Consider the *set-level* heuristic in the context of the robot example in previous section. 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 equivalent to the *set-level* heuristic extracted from a planning graph without mutex constraints. 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 the planning graph, we are able to detect that the robot cannot be at any square other than its current square.

Tables 1 and 2 show that the *set-level* heuristic outperform HSP-r and is competitive with Graphplan[5] in domains such as grid, mystery, travel and 8-puzzle[6] 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 in which the *sum* heuristic typically does well such as the rocket world and logistics. These are domains where the subgoals are fairly independent of each other. Closely examining the heuristic values reveals that the *set-level* heuristic is 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 idea. Specifically, it estimates the cost of a set in terms of the costs of its partitions. The second family, called "adjusted-sum" heuristics, attempts to improve the *sum* heuristic using the *set-level* idea. Specifically, it starts explicitly from the *sum* heuristic and then considers accounting for the interactions among subgoals based on the planning graph's level information. These two families are described separately in the next two subsections.

4.2. Partition-k heuristics

When the subgoals are relatively independent, the sum of their cost gives a much better estimate, whereas the level value of the set tends to underestimate cost 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: $S = S_1 \cup S_2 \cup \cdots \cup S_m$ (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. Naturally, we are interested in a partitioning in which the propositions in different partitions do not interact among each other. By *interacting* we mean the two propositions have either a of dynamic (level specific) or a static mutex in the planning graph. These notions are formalized by the following definitions.

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

[6] 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.

Table 1
Number of actions/total CPU time in seconds. The dash (–) indicates that no solution was found in 3 hours or 500 MB. 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	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
8puzzle-1	31/2444.22	33/196.73	31/4658.87	35/80.05	39/130.44	39/78.36	39/119.54	31/143.75
8puzzle-2	30/1545.66	42/224.15	30/2411.21	38/96.50	36/145.29	42/103.70	48/50.45	30/348.27
8puzzle-3	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 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

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
8puzzle-1	–	1078/603	99983/56922	1343/776	1114/613	1079/603	1980/1129	2268/1337
8puzzle-2	–	1399/828	51561/29176	1575/926	1252/686	1540/899	475/279	6544/3754
8puzzle-3	–	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

Definition 3. The (*binary*) *interaction degree* between two propositions p_1 and p_2 is defined as

$$\delta(p_1, p_2) = lev\big(\{p_1, p_2\}\big) - \max\big\{lev(p_1), lev(p_2)\big\}.$$

For any pair of propositions p_1, p_2 that are present in the planning graph, it is simple to note that when p_1 and p_2 are dynamic mutex, $0 < \delta(p_1, p_2) < \infty$. When p_1 and p_2 are static mutex, $\delta(p_1, p_2) = \infty$. When p_1 and p_2 have no mutex relation, $\delta(p_1, p_2) = 0$.

Definition 4. Two propositions p and q are *interacting* with each other if and only if $\delta(p, q) > 0$. Two sets of propositions S_1 and S_2 are non-interacting 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). *$h_{part-k}(S) \leftarrow \sum_{S_i} lev(S_i)$, where $S_1, \ldots, S_m$ are k-sized partitions of S.*

The question of how to partition the set S when $1 < k < |S|$, however, is interesting. For example, for $k = 1$, we have $h(S) = \sum_{p \in S} lev(p)$. From Table 1, we see that the *partition*-1 heuristic exhibits similar behavior to *sum mutex* heuristic in domains where the subgoals are fairly independent (e.g., gripper, logistics, rocket), and it is clearly better than the *sum mutex* heuristic in all other domains except blocks world.

For $k = |S|$, we have the *set-level* heuristic, which is very good in a set of domains that the *sum mutex* heuristic does not perform very well (e.g., grid world, travel, mprime).

For $k = 2$, we can implement a simple greedy pairwise partitioning scheme as follows: (i) Choose the pair of propositions p_1, p_2 in S that gives the maximum interaction degree $\delta(p_1, p_2)$ (if there are more than one such pair, simply choose one randomly), make $\{p_1, p_2\}$ one of the partitions of S. (ii) Now recursively partition the set $S - \{p_1, p_2\}$ using the same approach.[7]

As Table 1 shows, the resulting heuristic exhibits interesting behavior: It can solve many problems that are intractable for either 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-step solution for bw-large-d (19 blocks) after generating just 5328 nodes and expanding 925 nodes in 2423 sec.

A related question is which value of k leads to the best partition-k heuristic. 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.

[7] We note that this greedy scheme may not (and is not intended to) give a 2-partition with the highest cumulative δ values of all the partitions.

Adaptive partitioning. Another attractive idea is to consider "adaptive partition" heuristics that do not insist on equal sized partitions. Based on our definition of subgoal interaction (Definition 4), 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.

4.3. Adjusted-sum heuristics

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

Let $cost(p)$ denote the cost of achieving a proposition p according to the *sum* heuristic. Note that 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 $S = \{p_1, p_2, \ldots, p_n\}$. Suppose $lev(p_1) \leqslant lev(p_2) \leqslant \cdots \leqslant lev(p_n)$. Under the assumption that all propositions are independent, we have

$$cost(S) \leftarrow cost(S - p_1) + cost(p_1).$$

Since $lev(p_1) \leqslant lev(S - p_1)$, proposition p_1 is *possibly* achieved before the set $S - p_1$. Now, we assume that there are no positive interactions among the propositions, but there *may be* negative interactions. 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 as follows. 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 may be 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:

$$cost(S) \leftarrow cost(S - p_1) + cost(p_1) + \big(lev(S) - lev(S - p_1)\big). \tag{2}$$

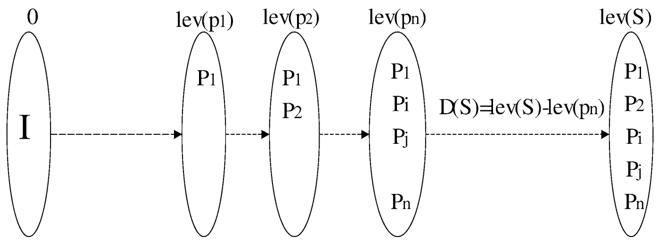

Fig. 4. The cost incurred for ignoring the negative interactions.

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

$$cost(S) \leftarrow \sum_{p_i \in S} cost(p_i) + lev(S) - lev(p_n).$$

Note that $lev(p_n) = \max_{p_i \in S} lev(p_i)$ as per our setup. Thus the estimate above is composed of the *sum* heuristic function $h_{sum} = \sum_{p_i \in S} cost(p_i)$ and an additional cost $lev(S) - \max_{p_i \in S} lev(p_i)$. We will call this difference the *interaction degree* among propositions in set S.

Definition 5. The *interaction degree* among propositions in a set S is

$$\Delta(S) = lev(S) - \max_{p \in S} lev(p).$$

It should be noted that the notion of binary interaction degree (Definition 2) is only a special case of the above definition for a set of two propositions. In addition, when there is no negative interaction among subgoals, $\Delta(S) = 0$, as expected (see Fig. 4). We have the following heuristic:

Heuristic 6 (Adjusted-sum heuristic).

$$h_{adjsum}(S) \leftarrow \underbrace{\sum_{p_i \in S} cost(p_i)}_{h_{sum}(S)} + \Delta(S).$$

Tables 1 and 2 show that this heuristic does very well across all the different types of problems that we have considered. From Table 1, we see that the solutions provided by the adjusted-*sum* heuristic are longer than those provided by other heuristics in many problems. The reason for this is that in many domains achieving a subgoal also helps achieve others, and so the first term $h_{sum}(S) = \sum_{p_i \in S} cost(p_i)$ actually overestimates. We can improve the solution quality by replacing the first term of $h_{adjsum}(S)$ by another estimate, $cost_p(S)$, that takes into account this type of *positive* interaction 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 a proposition in S such that $lev(p_S) =$

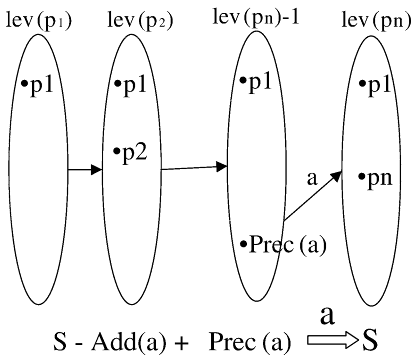

Fig. 5. Extracting the length of a plan that ignores the negative interactions.

$\max_{p_i \in S} lev(p_i)$. 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, note that none of them will be noop actions, and we would select one randomly.)

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

$$cost_p(S) \leftarrow 1 + cost_p\big(S + Prec(a_S) - Add(a_S)\big). \tag{3}$$

This regression accounts for the positive interactions 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 $lev(p_S)$ (because a_S achieve p_S at level $lev(p_S)$), recursively applying Eq. (3) to its right hand side will eventually reduce to a state S_0 where $lev(S_0) = 0$, and $cost_p(S_0)$ is 0 (see Fig. 5).

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 proposed in [5,18].

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). $h_{adjsum2}(S) \leftarrow cost_p(S) + \Delta(S)$ *where* $cost_p(S)$ *is computed using Eq.* (3)*, and* $\Delta(S)$ *is defined by Eq.* (5)*.*

Table 1 shows that *adjusted-sum2* heuristic can solve all types of problems considered. The heuristic is only slightly worse than the *adjusted-sum* heuristic in terms 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 often gives optimal or near optimal solutions.

Finally, if we discard the third term in the *adjusted-sum* heuristic, we would get an estimate consisting of two two components, $h_{sum}(S)$, 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 sum of these two different heuristic estimates yields a combination of *differential* power effective in wider range of problems.

Heuristic 8 (Combo heuristic). *$h(S) \leftarrow h_{sum}(S) + h_{lev}(S)$, 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 the *adjusted-sum* heuristic across all types of problems while the solution quality remains comparable.

4.4. Further improvements to the adjusted-sum heuristic functions

Since the family of *adjusted-sum* heuristics turns out to be the most effective, we devote this subsection to further analysis of and possible avenues of improvement for this heuristic family. 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 account for the positive interactions while ignoring the negative interactions, and $\Delta(S)$, which intends to measure the “penalty” cost for ignoring such negative interactions.

Improving the informedness of the *adjusted-sum* heuristic family involves a more accurate account of the negative interactions captured by $\Delta(S) = lev(S) - \max_{p_i \in S} lev(p_i)$, i.e., increasing its value. One way to do this is to improve the accuracy of $lev(S)$. Recall that $lev(S)$, by definition, is the index of the first level in the planning graph in which all propositions in S are present and not mutex with one another. Following Proposition 1, improving the accuracy of $lev(S)$ involves computing and propagating more mutexes, including higher order ones. Unfortunately, this task is also computationally hard.[8] 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 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 $S = \{p_1, p_2, \ldots, p_n\}$. For each pair of p and q, the interaction is reflected by its *interaction degree*, as defined 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:

$$\Delta_{max}(S) = \max_{p,q \in S} \delta(p, q). \tag{4}$$

Heuristic 9 (Adjusted-sum2M heuristic). *$h_{adjsum2M}(S) \leftarrow cost_p(S) + \Delta_{max}(S)$ where $cost_p(S)$ is computed using Eq.* (3)*, and $\Delta(S)$ is as given in Eq.* (4)*.*

[8] Even computing all binary mutexes can be as hard as the planning problem [2].

Table 3
Evaluating the relative effectiveness of the *adjusted-sum2* and *adjusted-sum2M* heuristics 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 plan found and the total run time in seconds. #N-gen and #N-exp are number of nodes generated and expanded during the search, respectively

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

It can be shown that when only binary mutexes are computed in the planning graph, $\Delta_{max}(S) \geqslant \Delta(S)$.[9] 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 $\Delta_{max}(S) \geqslant \Delta(S)$ for all S, $h_{adjsum2M}$ tends 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 of 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,

[9] 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, $\Delta_{max}(S) \geqslant \delta(p_m, p_r) = lev(\{p_m, p_r\}) - \max\{lev(p_m), lev(p_r)\} = lev(S) - \max\{lev(p_m), lev(p_r)\} \geqslant lev(S) - \max_{p_i \in S} lev(p_i) = \Delta(S)$.

we might want to partition the set S into pairs $S_i = \{p_{i1}, p_{i2}\}$ 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 experiments however, the *adjusted-sum2* heuristic using Δ_{sum} did not result in any significant improvement over *adjusted-sum2M* heuristic on most problems. It actually turned out to be worse in several domains. The reason, we think, is that in most domains, the independence among interactions themselves is not a reasonable assumption to make.

We conclude this section with a brief discussion of avenues for improving the admissibility of the *adjusted-sum* heuristics. The first component in the *adjusted-sum* heuristic function, $cost_p(S)$, 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 for the original problem. 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 [4]. 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 of course make the computation much more costly.

5. Finding optimal plans with admissible heuristics

We now focus on admissible heuristics that can be used to produce optimal plans (where the plan cost is measured in terms of the number of actions in the plan). While research in the AI search literature concentrates entirely on admissible heuristics (and finding optimal solutions), traditionally, efficient generation of optimal plans has received little attention in the planning community. In [20] Kambhampati et al. point out that while standard Graphplan is not guaranteed to produce optimal plans in terms of number of actions, [10] a version called "serial Graphplan", which uses a serial planning graph (see Section 3) instead of the standard planning graph, is guaranteed to produce optimal plans. In contrast, none of the known heuristic state space planners [3,4,33,41] focus on generating optimal solutions efficiently. [11]

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 [4], Bonet and Geffner

[10] Standard Graphplan algorithm produces plans that are optimal in the number of steps. Since a step can contain an arbitrary number of actions, it is easy to create problems where Graphplan produces a k-step plan with m actions, even when there is an n-action serial plan ($k < n < m$). For serial Graphplan, each step contains exactly one action, and thus step optimality implies optimality with respect to number of actions.

[11] An exception is the work by Haslum and Geffner [17], which was done around the same time as ours; see Section 9 for a discussion.

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

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	–	–

introduced the *max heuristic* that is admissible. In the previous section, we introduced the *set-level* heuristic that is admissible and gives a 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 the *set-level* heuristic is significantly more effective than the *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. Domains such as logistics and gripper remain intractable under reasonable time and memory limits.

The main problem once again is that the *set-level* heuristic still hugely underestimates the cost of a set of propositions. One reason for this is that there are many *n-ary* ($n > 2$) *dynamic* mutex constraints implicitly present in the planning graph, that are never marked during planning graph construction, and thus cannot be used by the *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 in the planning graph. The mutex computation and propagation corresponds to computing the *set-level* value $lev(S)$ for $|S| \leqslant k$. For $|S| > k$, the *set-level* heuristic value becomes:

$$h(S) \leftarrow lev(S) = \max_{p_1, p_2, \ldots, p_k \in S} lev\big(\{p_1, p_2, \ldots, p_k\}\big). \tag{5}$$

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, exhaustively propagating all higher order mutexes is likely to be an infeasible idea [2,20], as it essentially amounts to full consistency enforcement of the underlying CSP. According to Blum [2], 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 [22], to detect higher order mutexes in the form of "memos". 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 the next 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 8puzzle-2, the normal *set-level* 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 on all problems we considered (up to $3\times$ in the 8puzzle-2, $6\times$ 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 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.

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 the 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 interaction 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, as it has to store many more mutex constraints. The blocks world or grid world are examples of such 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 size may also become unwieldy, exhausting the available memory in some cases. For instance, the program runs out of memory (250 MB) on problem bw-large-d when the *combo* heuristic is extracted from a normal planning graph.

Fortunately, there are a variety of techniques for improving the efficiency of planning graph construction in terms of both time and space. We discuss two important ones below.

Using compact representations of planning graphs. An obvious optimization is the use of compact bi-level representation that exploits the structural redundancy in the planning graphs (cf. STAN planner [30]), which can help improve the cost of heuristic computation significantly. In fact, in one of our experiments, by using a bi-level planning graph as a basis

Table 5
Total CPU time improvement from using bi-level planning graphs for computing *combo* heuristic

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

for our heuristics,[12] we could achieve significant speedups (up to 7×) in all problems, and we were also able to solve more problems than before because the planning graph takes less memory (see Table 5).

Using partial planning graphs. We can also limit the heuristic computation cost more aggressively by trading heuristic quality for reduced cost of the planning graph's construction. Specifically, in the previous sections, we discussed the extraction of heuristics from a leveled planning graph. Since the state search does not operate on the planning graph directly, we need not use the full leveled graph to preserve completeness. *Any subgraph* of the full leveled planning graph can be utilized as the basis for the heuristic computation. There are at least three ways of computing a subgraph of the leveled planning graph:

(1) Grow the planning graph to some length less than the level where it levels off. For example, we may grow the graph up to the level where all the top level goals of the problem are present without any mutex relations.
(2) Limit the time spent on marking mutexes on the planning graph.
(3) Introduce only a subset of applicable actions at each level of the planning graph. For example, we can exploit techniques such as RIFO [35] and identify a subset

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

of the action instances in the domain that are likely to be relevant for solving the problem.

Any combination of these 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 of the heuristic are unaffected by the first two techniques.

We have implemented the first technique in a planner called *AltAlt* (see Section 7). Specifically, we build a compact bi-level planning graph only up to the level where all of the subgoals first appear without being mutex, i.e., level $lev(S)$, where S is the set of subgoals being considered. The rationale for doing this is that most relevant states regressed from S are likely to have the *lev* value less than $lev(S)$. As a result, the heuristic function value for these states will remain unchanged even if the planning graph is not expanded to level off.

When a graph is not grown to level off, the notion of *lev* has to be redefined as follows:

Definition 6 (*Level in a partially grown graph*). $lev{*}(S) = \min(lev(S), l + 1)$, where l is the index of the last level that the planning graph has been grown to.

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 7.1 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 planning graphs for computing the heuristics for guiding the state search, as mentioned earlier in Section 3, the planning graph is also used to pick the action instances considered in expanding the regression search tree. The set of actions occuring in the planning graph has two useful characteristics:

1. All action instances that are applicable in reachable states are present in the final level of a leveled planning graph.
2. In regression search, action instances that first occur in earlier levels are more likely to be applicable to states that are closer to 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. From the property 1 above, we can see that if the graph has been grown to level off, limiting regression search to this subset of actions is guaranteed to preserve completeness. Using just this set of reachable actions would reduce the branching factor for the regression search. In addition, when there is a set of actions A such that $f(Regress(S, a))$ has the same value for all $a \in A$, we could break tie by favoring an action in A that occurs in the planning graph the earliest.

A more aggressive, albeit theoretically incomplete, selective expansion approach, that we call *sel-exp* strategy, involves the following. To begin with, instead of growing the planning graph to level off, we expand it up to level $l = lev(G)$, where G is the set of initial subgoals. Suppose that we are trying to expand a state S during the regression search, then only the set of actions appearing in the action level $lev(S)$ is considered for regressing 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,[13] in practice we have not found a single instance in which *sel-exp* strategy fails to find a solution that can be found considering the full set of actions. As we shall see in the next section, the *sel-exp* strategy has a significant positive effect on the performance of *AltAlt* in some domains such as the Schedule World [1].

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

Most of the heuristic functions developed in the previous sections are based on computing the level value of some set of propositions. This function can also be computed using dynamic programming (DP) style computation without explicitly growing planning graphs. Specifically, according to Proposition 1, this computation boils down to computing *lev* function value for all pairs of propositions (assuming that only binary mutexes are computed). Mutex marking and propagation in the planning graph directly correspond to the updating procedure of the *lev* function. In addition, upon convergence *lev* values directly correspond to the *lev* values extracted from a leveled planning graph. In fact, in a recent work, Haslum and Geffner [17] start 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 a bottom-up DP approach.

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 to the DP style computation. We will show that this is not the case. In fact, using a bi-level representation of the graph and efficient mutex marking procedures such as those in STAN [30], the time taken for building the graph tends to be *lower* than the time taken for the updating procedures to converge. Furthermore, the additional amount of memory required to store such a graph is often not very significant and serves a useful purpose.

Indeed, the level-by-level expansion of the planning graph can be seen as a DP style updating procedures for *lev* values. The interesting point is that, only the actions that are applicable in the previous level are considered in the updating procedures, and only the 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 the updating process.

Let us now consider the issue of memory. Let P and A be the total number of (grounded) propositions and actions, respectively, for a given problem. Let α, β, and γ be the average number of bytes needed to store a single proposition, a mutex relation, and an action, respectively. Using a DP computation, only the *lev* values of all propositions and the pairs of propositions—there are $P(P-1)/2$ of those—are stored, taking $(\alpha P + \beta P(P-1)/2)$ bytes. In the bi-level representation of planning graph, the set of propositions, pairs of

[13] Because some actions needed for the solution plan that appear much later at levels of index greater than l are not considered.

mutexes, and the set of actions are all that need to be stored. These numbers are essentially independent of the number of 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 duplication as in the original Graphplan's planning graph. Thus, the amount of memory needed to store the bi-level graph structure is $(\alpha P + \beta P(P-1)/2 + \gamma A)$ bytes. The additional amount of memory the planning graph actually has to store is the set of all actions, which is γA bytes. This amount is often not very significant, as it is independent of the number of graph levels. Moreover, as we saw in Section 6.2, the set of actions is actually very helpful in focusing the search.

To summarize, contrary to initial impressions, the planning graph does not take much more memory than using DP procedures 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*, [14] a planner implemented in C programming language. *AltAlt* is implemented on top of two highly optimized existing planners—STAN [30] that is a very effective Graphplan style planner is used to generate planning graphs, and HSP-r [3], a heuristic search planner provides an optimized state search engine. *AltAlt* uses planning graph based heuristics developed in Section 4.4 to guide its search. Empirical results show that *AltAlt* can be orders of magnitude faster than both STAN and HSP-r, validating the effectiveness of the heuristics being used. An earlier version of *AltAlt* took part in the automated track in the 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 competitive with the fastest planners in the competition, including Hoffman's FF [18], Bonet and Geffner's HSP2.0 [3,4], and Refanidis's GRT [41].

The high-level architecture of *AltAlt* is shown in Fig. 6. 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 [30] for this purpose as it provides fast mutex marking routines and memory-efficient implementation of the planning graph (see below). This planning graph structure is then fed to a heuristic extractor module that is capable of extracting a

[14] $\underline{\text{A}}$ $\underline{\text{L}}$ittle of $\underline{\text{T}}$his and $\underline{\text{a}}$ $\underline{\text{L}}$ittle of $\underline{\text{T}}$hat.

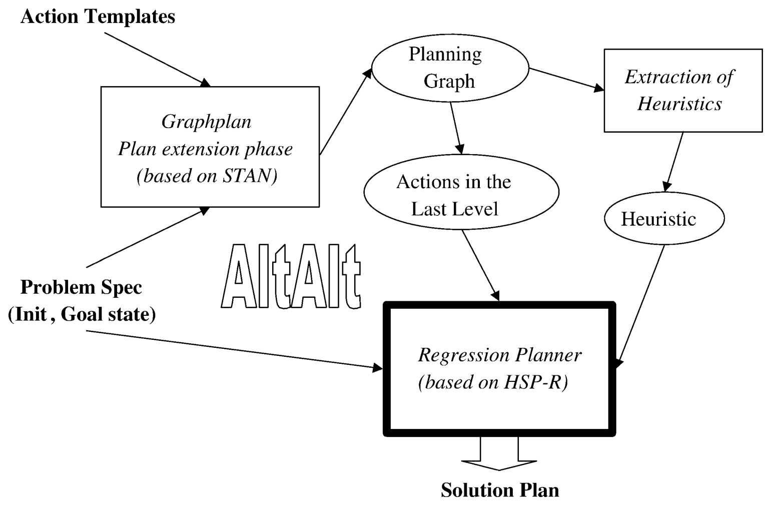

Fig. 6. Architecture of *AltAlt*.

variety of effective and admissible heuristics, described in Section 4. The heuristics, along with the problem specification, and the set of ground actions in the final action level of the planning graph structure (see below for explanation) are fed to a regression state-search planner. The regression planner code is adapted from HSP-r [3].

7.1. Evaluating the performance of AltAlt

AltAlt's performance on many benchmark problems, as well as the test suite used in the 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:

(1) *AltAlt* convincingly outperforms STAN and HSP-r, showing that it indeed leverages complementary strengths of Graphplan and heuristic state search planning.
(2) *AltAlt* is able to reduce the cost of its heuristic computation with very little negative impact on the quality of the solutions produced (see Section 7.2).

Our experiments were all done on a Linux system running on a 500 MHz pentium III CPU with 256 megabytes of RAM. We compared *AltAlt* with the latest versions of both STAN and HSP-r systems running on the same hardware. We also compared *AltAlt* to FF and HSP2.0, which are two of the best planners at the AIPS-2000 competition [1]. HSP2.0 is a recent variant of the HSP-r system that runs both regression search (HSP-r) and progression search (HSP) in parallel. FF is a very efficient state search planner using a novel local search 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, but also contain some other benchmark problems from the literature. Unless noted otherwise, in all the experiments, *AltAlt* was run with the heuristic $h_{AdjSum2M}$ (Section 4.4), 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 *sel-exp* strategy in Section 6.2).

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)

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

	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
8puzzle-1	37.40	31	34.47	45	0.64	59	0.19	47	0.69	31
8puzzle-2	35.92	30	6.07	52	0.55	48	0.15	84	0.74	30
8puzzle-3	0.63	20	164.27	24	0.34	34	0.08	42	0.19	20
8puzzle-4	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

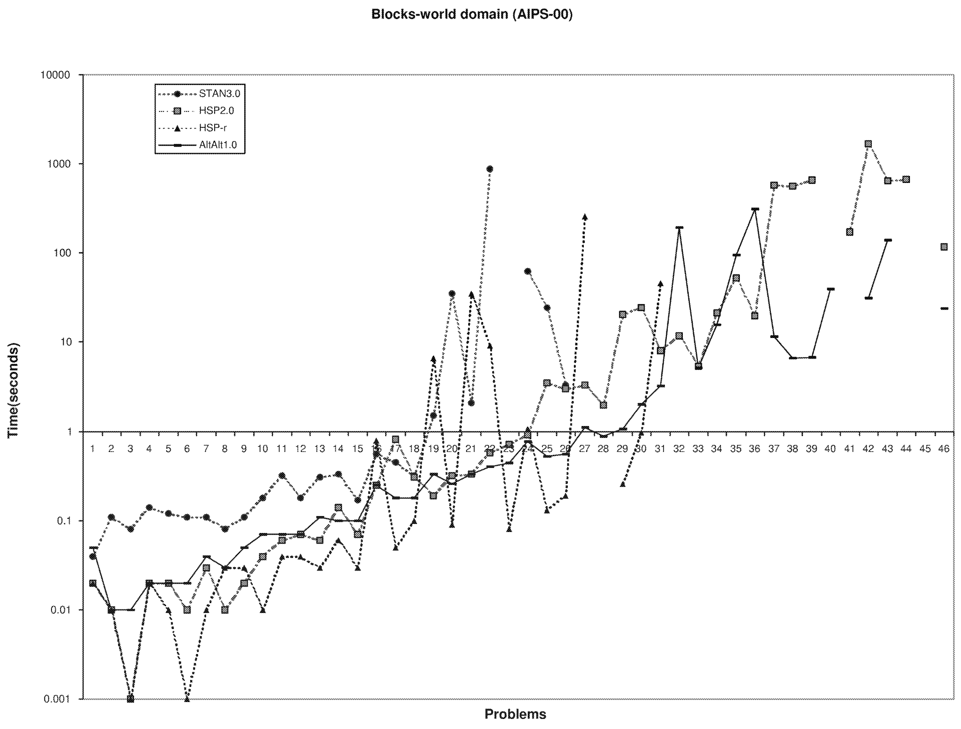

Fig. 7. Results in blocks world domain.

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 and those that are hard for HSP-r. We also note that the solutions produced by *AltAlt* are as good as or of better quality than those produced by the other two systems on 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 that *AltAlt* solves. *AltAlt* gives better solution quality in most problems.

The plots in Figs. 7, 8 and 9 compare the time performance of STAN, HSP-r, HSP2.0 and *AltAlt* in specific domains. The plot in Fig. 7 summarizes the problems from blocks world and the plot in Fig. 8 refers to the problems from logistics domain, while the plot in Fig. 9 refers to the problems from the scheduling world. These are three of the standard benchmark domains 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 competitive with it in the 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 in this domain. HSP-r is unable to solve any problem in this domain. Although not

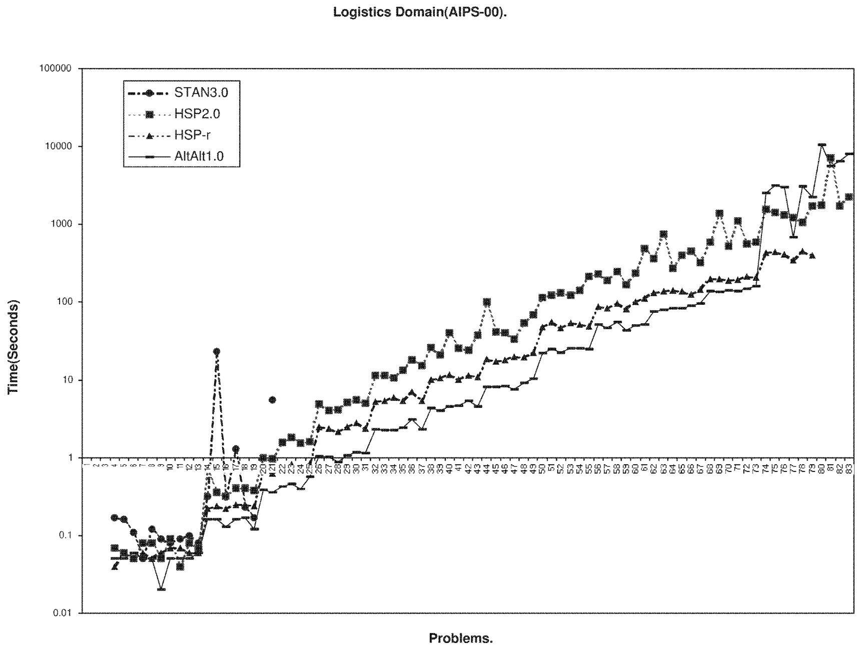

Fig. 8. Results in logistics domain.

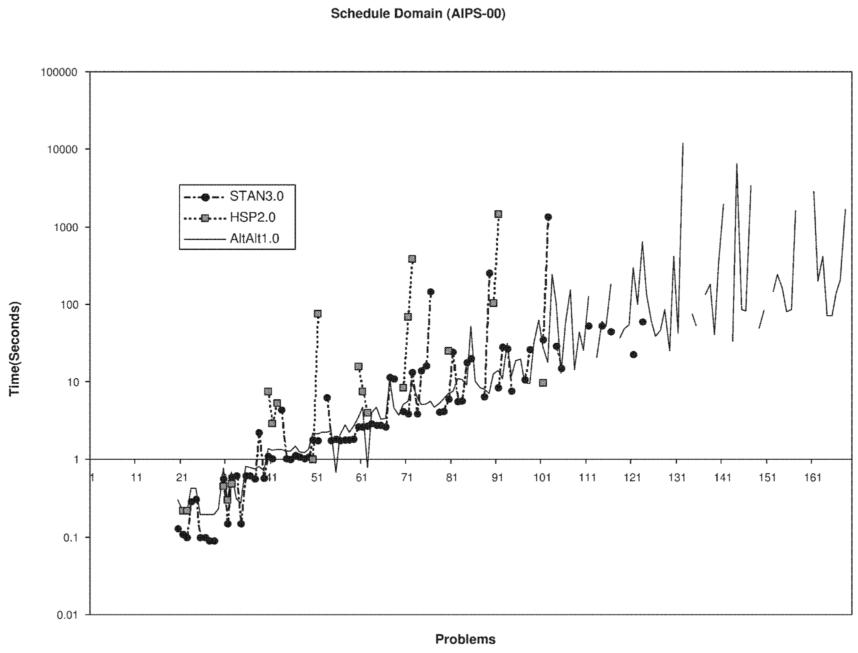

Fig. 9. Results in schedule domain.

shown in the plots, the length of the solutions found by *AltAlt* in all these domains was as good as, or better than, the other two systems.

7.2. Using partial graphs to reduce the heuristic computation cost

In the experiments described in the previous section 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 may trade cost of the heuristic computation for solution quality. To see if this is indeed the case, 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 Figs. 10 and 11 show the results of the experiments with a large set of problems from the scheduling domain. Fig. 10 shows the total time taken for heuristic computation and search, Fig. 11 compares the length of the solution found for both strategies. We can see very clearly that if we insist on full leveled planning graph, *AltAlt* is unable to solve problems beyond the first 81, while the heuristic derived from the partial planning graph scales all the way to 161 problems. The total planning time associated

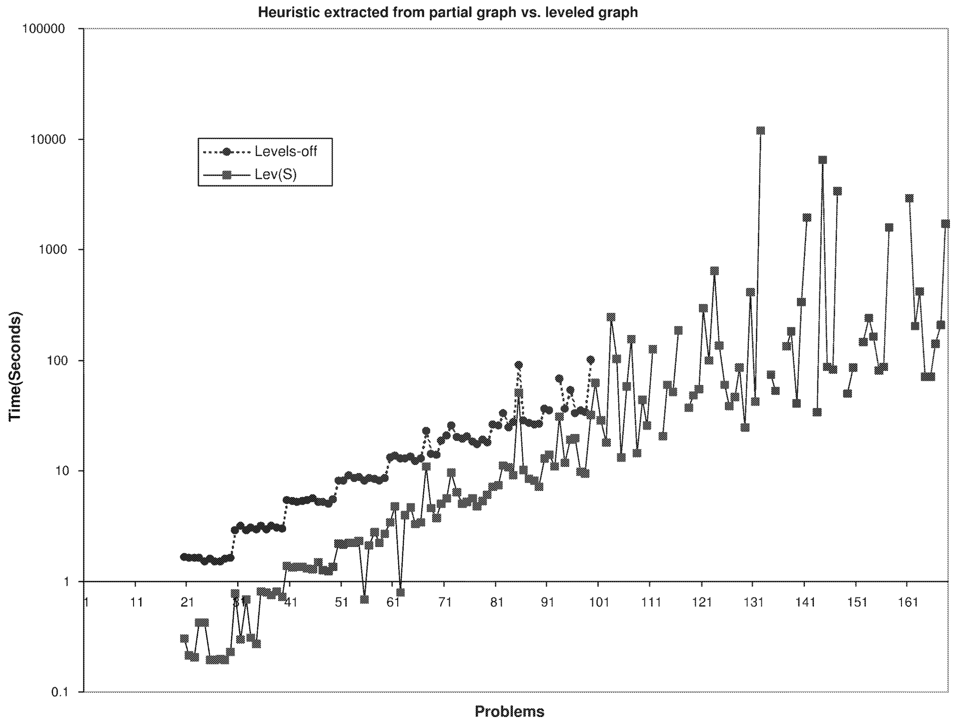

Fig. 10. Results on trading heuristic quality for cost by extracting heuristics from partial planning graphs. Comparison of running times.

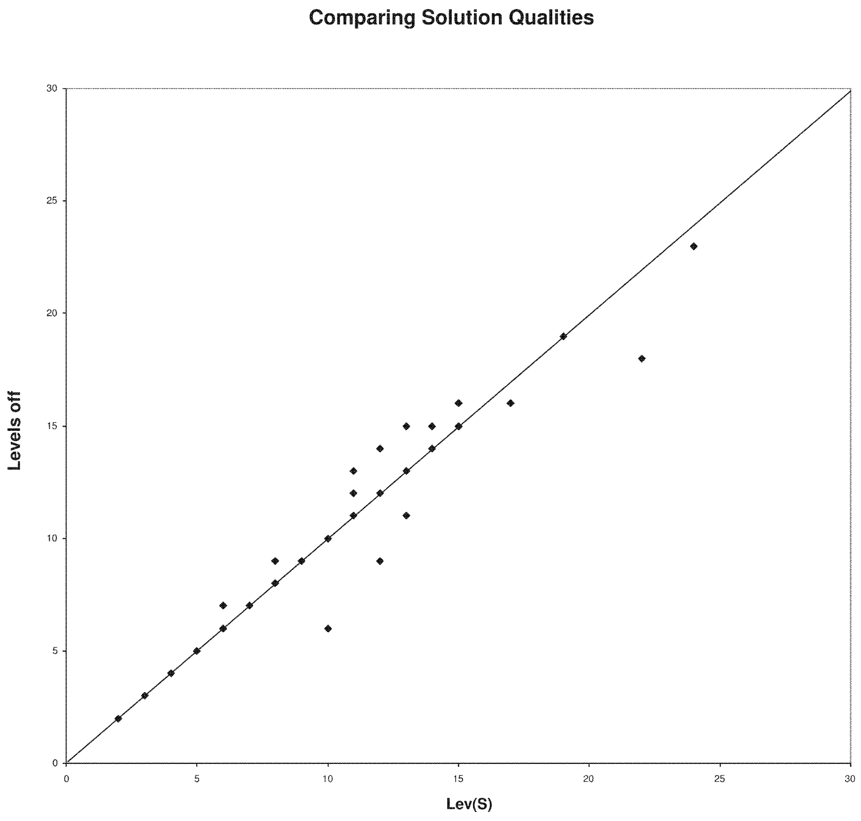

Fig. 11. Results on trading heuristic quality for cost by extracting heuristics from partial planning graphs. Comparison of the quality of solutions.

with the partial planning graph based heuristic is significantly lower, as expected, in all problems. The plot in Fig. 11 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. There are very few points below the diagonal corresponding to the problem instances on which the plans generated with the heuristic derived from the partial planning graph are longer than those generated with heuristic derived from the full leveled planning graph. Interestingly there are also instances where the solutions obtained using heuristics derived from partial planning graphs are actually better than those obtained using heuristics derived from the full leveled graph. We note that this behavior, while surprising, is nevertheless theoretically plausible, given that the $h_{AdjSum2M}$ heuristic is not admissible to begin with.

The results above validate our contention in Section 6.1 that the heuristic computation cost can be kept within limits. It should be mentioned here that the planning graph computation cost depends a lot upon domain. 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 from 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 the planning graph can also serve as the basis for extracting heuristics to improve Graphplan's own CSP style search. 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 the variable and value ordering strategies used by Graphplan 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, we evaluate the effectiveness of these heuristics and discuss the advantages they offer.

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 the 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 [20,24,45] had explicated the connections between this backward search phase of Graphplan algorithm and constraint satisfaction problems (specifically, the dynamic constraint satisfaction problems, as introduced in [34]). 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* (*goal*) *ordering* heuristic. The order in which the actions supporting a goal are considered for inclusion in the solution graph is the *value* (*action*) *ordering* heuristic. In their original paper [2], Blum and Furst argue that variable and value ordering heuristics are not particularly useful in improving Graphplan, mainly because exhaustive search is required in the levels before the solution bearing level anyway. There are however reasons to pursue variable and value ordering strategies for Graphplan.

(1) In many problems, the search done at the final level accounts for a significant part of the overall search. Thus, it will be useful to pursue value and variable ordering strategies, even if they improve only the final level search.
(2) 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 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 the planning graph using some of the same ideas used on deriving the state search heuristics (in Sections 4 and 5). 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,[15] it turns out that it is hardly infallible. Our experiments with Graphplan reported in [23] showed that using noops first heuristic can, in many domains, drastically worsen the performance. Specifically, in most of the problems, considering noops first worsens 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 the CSP literature, the standard heuristic for variable ordering involves trying the most constrained variables first [44]. A variable is considered most constrained if it has the least number of actions supporting it. Although some implementations of Graphplan such as SGP [45] 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 [19] show that the most constrained first heuristic leads to about 4$\times$ speedup at most.

One reason for the ineffectiveness of the 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

[15] In fact, some of the extensions of Graphplan search, such as Koehler's incremental goal sets idea [25] explicitly depend on Graphplan using noops first heuristic.

assignment for that variable. This is because 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.

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 estimation.

To make this connection clearer, let us consider the planning graph shown in the left of Fig. 12, and the corresponding CSP style backward search in 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 g from the initial state, which can be captured by the notion of level value of g, namely $lev(g)$ (see Definition 2).

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-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 $Prec(a_5) = \{p_4\}$. We can apply the family of heuristic functions

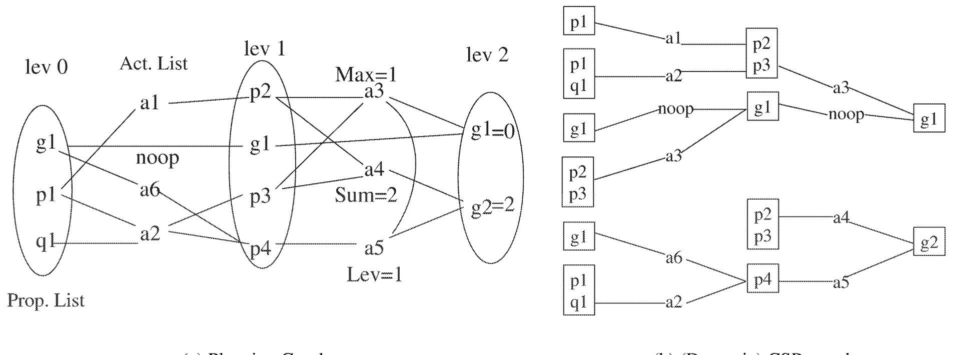

Fig. 12. A planning graph example and the CSP style regression search on it. To avoid cluttering, we do not show all the noops and mutex relations.

developed earlier for measuring the cost of achieving a set of preconditions for a given action.

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

> Propositions are ordered for assignment 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 the 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 from the level information extracted from the planning graph and thus are very cheap to compute.

Max heuristic: The cost of 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 Fig. 12 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 Fig. 12 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 the entire set of that action's preconditions are present without being mutex. Thus $cost_{Level}(a) = lev(Prec(a))$. For example, the cost of A_5 supporting G_2 in Fig. 12 is 1 because A_5 has one precondition P_4, and it occurs in level 1 of the planning graph for the first time. 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 the *Level* heuristic as using the uniform notion of "first appearance 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 $P_1, \ldots, P_{10}$, 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: $cost_{Max}(a) \leqslant cost_{Sum}(a)$ and $cost_{Max}(a) \leqslant cost_{Level}(a)$, 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. Note that all the heuristics use the same variable (goal) ordering strategy, and differ only in the way they order values (actions). Our implementations were based on the Graphplan implementation bundled in the Blackbox system [24], which in turn was derived from Blum and Furst's original implementation [2] 56. 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 value ordering (and the same "highest level first" idea for variable ordering). CPU time is shown in minutes. For our Pentium Linux machine 500 MHz with 256 MB 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).

Table 7
Effectiveness of *Level* heuristic in solution bearing planning graphs. The columns titled Lev 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 Lev 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 pairwise mutex

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	0.008	12/12	0.005	12/12	0.005	12/12	0.006	1.6×	1.6×	1.3×
bw-large-b	18/18	0.76	18/18	0.13	18/18	0.13	18/18	0.085	5.8×	5.8×	8.9×
bw-large-c	–	>30	28/28	1.15	28/28	1.11	–	>30	>26×	>27×	–
huge-fct	18/18	1.88	18/18	0.012	18/18	0.011	18/18	0.024	156×	171×	78×
bw-prob04	–	>30	8/18	5.96	8/18	8	8/19	7.25	>5×	>3.7×	>4.6×
rocket-ext-a	7/30	1.51	7/27	0.89	7/27	0.69	7/31	0.33	1.70×	2.1×	4.5×
rocket-ext-b	–	>30	7/29	0.003	7/29	0.006	7/29	0.01	10000×	5000×	3000×
att-log-a	–	>30	11/56	10.21	11/56	9.9	11/56	10.66	>3×	>3×	>2.8×
gripper-6	11/17	0.076	11/15	0.002	11/15	0.003	11/17	0.002	38×	25×	38×
gripper-8	–	>30	15/21	0.30	15/21	0.39	15/23	0.32	>100×	>80	>93×
ferry41	27/27	0.66	27/27	0.34	27/27	0.33	27/27	0.35	1.94×	2×	1.8×
ferry-5	–	>30	33/31	0.60	33/31	0.61	33/31	0.62	>50×	>50×	>48×
tower-5	31/31	0.67	31/31	0.89	31/31	0.89	31/31	0.91	0.75×	0.75×	0.73×

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

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	0.008	12/12	0.006	12/12	0.006	12/12	0.006	1.33×	1.33×	1.33×
bw-large-b	18/18	0.76	18/18	0.21	18/18	0.19	18/18	0.15	3.62×	4×	5×
huge-fct	18/18	1.73	18/18	0.32	18/18	0.32	18/18	0.33	5.41×	5.41×	5.3×
bw-prob04	8/20	30	8/18	6.43	8/18	7.35	8/19	4.61	4.67×	4.08×	6.5×
rocket-ext-a	7/30	1.47	7/26	0.98	7/27	1	7/31	0.62	1.5×	1.47×	2.3×
rocket-ext-b	–	>30	7/28	0.29	7/29	0.29	7/28	0.31	>100×	>100×	>96×
tower-5	31/31	0.63	31/31	0.90	31/31	0.89	31/31	0.88	0.70×	0.70×	0.71×

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 in this case. This suggests that a large proportion of the search time is spent on the failing levels, and the current combination of our value ordering and variable ordering heuristics generally do not help detect failures early in these levels.

To understand the relative roles played by value (action) and variable (goal) ordering strategies in improving the efficiency of Graphplan search, we also ran the Graphplan armed with only variable ordering heuristic, and Graphplan armed with only value ordering heuristic (which is the *max* heuristic). In all cases, we consider search in the first solution bearing level. The results are shown in Table 9. They show that variable ordering alone is less effective than value ordering heuristic.

Interestingly, the results in Table 9 also show that the variable ordering heuristic when *combined* with the value ordering *max* heuristic provides even better performance than value ordering heuristic alone, in a number of problems, most notably those in the blocks world domain. In this domain, the improvement ranges from 5 up to several orders of magnitude. This type of synergy between variable and value ordering heuristics, while not entirely surprising, is quite desirable. Generally, variable ordering heuristics attempt to improve performance in failing branches, while the value ordering heuristics aim to direct the search towards branches leading to solutions. Since the value ordering heuristics are rarely perfect, in that they can guarantee backtrack free search, there are always failing branches even with a value ordering heuristic. A synergistic variable ordering heuristic can improve performance on these branches.

Table 9
Results comparing the relative importance of the variable and value ordering heuristics on Graphplan's search (in the first solution bearing level)

Problem	Normal GP		Max GP		Var.Order only		Value.Order only	
	Length	Time	Length	Time	Length	Time	Length	Time
bw-large-a	12/12	0.008	12/12	0.005	12/12	0.010	12/12	0.007
bw-large-b	18/18	0.76	18/18	0.13	18/18	0.29	18/18	0.61
bw-large-C	–	>30	28/28	1.15	–	>30	–	>30
huge-fct	18/18	1.88	18/18	0.012	18/18	0.90	18/18	2.77
bw-prob04	–	> 30	8/18	5.96	–	>30	8/19	8
rocket-ext-a	7/30	1.51	7/27	0.89	7/30	2.75	7/27	0.35
rocket-ext-b	–	> 30	7/29	0.003	7/26	1.80	7/29	0.009
att-log-a	–	>30	11/56	10.21	–	>30	11/55	1.01
gripper-6	11/17	0.076	11/17	0.002	11/17	0.080	11/15	0.002
gripper-8	–	> 30	15/23	0.30	–	> 30	15/21	0.28
ferry41	27/27	0.66	27/27	0.34	27/27	0.87	27/27	0.24
ferry-5	–	> 30	31/31	0.60	–	> 30	31/31	0.46
tower-5	31/31	0.67	31/31	0.89	31/31	0.79	31/31	0.78

8.5. *Graph-length insensitivity of level based heuristics*

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.

Table 10 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 11 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

Table 10
Results showing that level based heuristics are insensitive to the length of the planning graph being searched

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	0.008	12/12	0.005	12/12	0.007	12/12	0.008	12/12	0.01
bw-large-b	18/18	0.76	18/18	0.13	18/18	0.21	18/18	0.21	18/18	0.25
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	0.012	18/18	0.01	18/18	0.02	18/18	0.02
bw-prob04	–	> 30	8/18	5.96	–	>30	–	>30	–	>30
rocket-ext-a	7/30	1.51	7/27	0.89	8/29	0.006	8/29	0.007	8/29	0.009
rocket-ext-b	–	> 30	7/29	0.003	9/32	0.01	9/32	0.01	9/32	0.01
att-log-a	–	>30	11/56	10.21	13/56	8.63	13/56	8.43	13/56	8.58
gripper-6	11/17	0.076	11/17	0.002	11/17	0.003	11/17	0.003	11/15	0.004
gripper-8	–	> 30	15/23	0.30	15/23	0.38	15/23	0.57	15/23	0.33
ferry41	27/27	0.66	27/27	0.34	27/27	0.30	27/27	0.43	27/27	0.050
ferry-5	–	> 30	33/31	0.60	31/31	0.60	31/31	0.60	31/31	0.61
tower-5	31/31	0.67	31/31	0.89	31/31	0.91	31/31	0.91	31/31	0.92

Table 11
Performance of *Level* and *Sum* heuristics in searching longer planning graphs

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	0.007	12/12	0.008	12/12	0.01	12/12	0.007	12/12	0.008	12/12	0.008
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	0.014	18/18	0.015	18/18	0.019	18/18	0.014	18/18	0.015	18/18	0.019
bw-prob04	11/18	18.88	–	>30	–	>30	–	>30	–	>30	–	>30
rocket-ext-a	9/28	0.019	9/28	0.02	9/28	0.02	8/28	0.003	8/28	0.004	8/28	0.006
rocket-ext-b	9/32	0.007	9/32	0.006	9/32	0.01	7/28	0.011	7/28	0.012	7/28	0.014
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	0.003	11/15	0.003	11/15	0.003	11/15	0.004	11/15	0.004	11/15	0.004
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

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.)

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.

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. 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 planning graphs. However, as the results in [23] 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

This section is organized into two parts. The first part (Section 9.1) deals with relations between our work and the existing work on heuristic search planners. The second part (Section 9.2) discusses the broad relations between our work and the work on heuristics in the AI search literature.

9.1. Relations to other heuristic search planners

Given the current popularity of heuristic search planners, it is somewhat surprising to note that the interest in the distance based heuristics in AI planning is a relatively new development. Ghallab and his colleagues were the first to report on a reachability based heuristic in IxTeT [14] for doing action selection in a partial order planner. However the effectiveness of their heuristic was not adequately established. Subsequently the idea of distance based heuristics was independently (re)discovered by McDermott [31,33] in the context of his UNPOP planner. UNPOP was one of the first domain-independent planners to synthesize 40 action long plans (Graphplan was the other, more well known, planner that showed such scale-up). A second independent re-discovery of the idea of using distance based heuristics in planning happened with Bonet and Geffner's work [4]. The respectable performance of HSP planner in the 1998 AIPS planning competition (where all the other competitors were variants of Graphplan algorithm) increased the community's interest in heuristic search planners in general. Since then, there have been a slew of efforts in developing effective heuristics for domain independent planners.

The current work on heuristics for planning can be classified broadly along three dimensions:

(1) the basis for the heuristics,
(2) the approach used to compute them, and
(3) the types of planners that the heuristics are directed towards.

We will briefly discuss these three dimensions and how the existing systems vary along them.

9.1.1. Basis for the heuristics

All current heuristic search planners derive their heuristics from reachability (distance) information. Informally, reachability information characterizes (sets) of world states in terms of their distance from the initial state. It should be clear that computing fully accurate reachability information for the entire state space is going to be harder than solving the planning problem in the first place. Thus, the heuristics are ofen based on coarse-grained (approximate) reachability information. Main variation among planners based on distance heuristics is in terms of the accuracy and granularity of the reachability information used. This in turn depends on the extent to which the subgoal interactions are accounted for in deriving the heuristics. Early planners such as UNPOP and HSP derived reachability information (and heuristics) with an assumption of subgoal independence. In other words, the distance of a state S containing the literals $p_1, p_2, \ldots, p_i$ from the initial state is seen as the sum of the distances of the individual literals. The informedness of these heuristics can be improved by accounting for negative interactions between subgoals, while their admissibility can be improved by accounting for the positive interactions.

Our work describes a way of using the planning graphs to aggressively accounting for both positive and negative interactions among subgoals. Specifically, our best heuristics such as *adjusted-sum2*, *adjusted-sum2M* and *combo* start with a heuristic estimate that uses the assumption of subgoal independence, but adjust this estimate to take both positive and negative interactions into account. We also discuss ways in which the accuracy of the heuristics can be improved in a graded way by improving the degree to which we account for the subgoal interactions (see Section 4.4).

Although there have been other recent planners that have considered accounting for either negative or positive interactions between subgoals, *AltAlt* is the first planner to support consideration of *both* types of interactions in a systematic way. It is nevertheless worth comparing *AltAlt*'s planning graph based approach to these other planners. We have already noted (Section 2.1) that HSP-r accounts for negative interactions using a form of static mutex relations between literals. *AltAlt*'s use of planning graphs allows it to exploit not just static (persistent) mutexes, but also dynamic (level specific) mutexes. We have argued that level specific mutexes improve the informedness of the heuristics, and our empirical comparisons with HSP-r validate this argument. As can be seen from the comparisons between *AltAlt* and HSP-r, the planing graph provides a more effective method for accounting for negative interactions.

FF [18] and GRT [41] focus on accounting for positive interactions between subgoals. Specifically, Hoffman [18] uses the length of the first relaxed plan found in a relaxed planning graph (without mutex computation) as the heuristic value. This can be seen as a special case of our *adjusted-sum2* heuristic $h(S) = cost_p(S) + \Delta(S)$, with $\Delta(S) = 0$, completely ignoring the negative interactions.[16] Refanidis [41] extracts the co-achievement relation among subgoals from the first relaxed plan to account for the positive interactions.[17]

Our work also reveals interesting connections between the extraction of distance based heuristics that do a better job of accounting for subgoal interactions, and the idea of improving the consistency level of a CSP encoding of the planning problem. Specifically, as has been noted elsewhere (cf. [9,19]) a planning graph can be seen as a CSP encoding whose solution corresponds to a valid plan for the problem. The propagation of mutex constraints on the planning graph can be seen as improving the consistency level of the underlying CSP encoding [8,20]. In particular, the Graphplan's standard mutex propagation routines can be seen as a form of directed partial 1- and 2-consistency enforcement [8, 20]. The fact that the informedness of the heuristics derived from the planning graph improves with mutex propagation, can be seen as explicating the connection between the effectiveness of the heuristics and the degree of consistency of the underlying (CSP) encoding from which they are derived. This connection is exploited in an interesting way in Section 5, where we described a way of improving the informedness of the *set-level* heuristic with the help of memos unearthed by a limited run of Graphplan's backward search. Memos can be seen as enforcing higher order consistency on the planning graph, and thus their use improves the heuristic. Finally, the invariants derived by domain pre-processors such as TIM [13] and DISCOPLAN [15], or in [42], can be seen as persistent (static) memos, and thus they too can improve the quality of the heuristics.

9.1.2. Computation of the heuristics

IxTET and UNPOP compute the distance heuristics on demand using a top-down procedure. Given a state S whose distance needs to be estimated, both IxTET and UNPOP do an approximate version of regression search (called "greedy regression graphs" in UNPOP) to estimate cost of that state. The approximation involves making an independence assumption and computing the cost of a state S as the sum of the costs of the individual literals comprising S. When a literal p is regressed using an action a to get a set of new subgoals S', S' is again split into its constituent literals. UNPOP also has the ability to estimate the cost of partially instantiated states (i.e., states whose literals contains variables). HSP and HSP-r use a bottom-up approach for computing the heuristics. They use an iterative fixed point computation to estimate the distance of every literal from the given initial state. The computation starts by setting the distance of the literals appearing in the initial state to 0, and the distance of the rest of the literals to

[16] Bonet and Geffner [5] also discussed a similar idea, but did not actually use it in their HSP planner.

[17] In fact, the extraction of the co-achievement relation among subgoals does consider the delete effects of the actions. However, this is not considered to the extent to adequately account for the negative interactions among the subgoals (because the first relaxed plan corresponding to which the co-achievement relations are extracted may or may not be the correct plan to be found).

∞. The distances are then updated until fixpoint using action application. The updating scheme uses independence assumptions in the following way: if there is an action a that gives a literal p, and a requires the preconditions $p_1, p_2, \ldots, p_i$, then the cost of achieving p is updated to be the minimum of its current cost, and the sum of cost of a plus the sum of the costs of achieving $p_1, p_2, \ldots, p_i$. Once this updating is done to fixed point, estimating the cost of a state S involves simply looking up the distances of the literals comprising S and summing them up. In their recent work, Haslum and Geffner [17] point out that this bottom up fixed point computation can be seen as an approximate dynamic programming procedure for computing the distances of the states from the initial state. The approximation involves setting the distance of a state S to the maximum of the distances of its k-sized subsets. With $k = 2$, this becomes equivalent to our *set-level* heuristic.

AltAlt uses the planning graph data structure to estimate the costs of the various states. Since the planning graph is computed once for a given problem (initial state), it is akin to a bottom-up computation of the distance heuristics. As we discussed in detail in Section 6.3, basing the bottom-up computation on the planning graph rather than on an explicit dynamic programming computation can offer some advantages: We have seen that *AltAlt* can use the information in the planning graph to derive a many qualitatively different heuristics over and above the "*sum*" heuristic. We have also noted (see Section 6.3) that the planning graph structure offers the important side benefit of action selection heuristics.

9.1.3. Types of planners using the distance heuristics

Distance based heuristics can be used to control the search of any type of classical planner. UNPOP and HSP both used their distance heuristics to control the search of a progression planner. A problem with the use of distance based heuristics to progression planners is that since the initial state changes from action application, the distance computation has to be repeated, and distance estimates updated, after every action application. This tends to increase the cost of heuristic computation. Two approaches have been explored to keep the cost of heuristic computation in check: (i) reversing the direction of distance computation and (ii) reversing the direction of search (refinement). The problem with reversing the direction of distance computation is that the goal state is rarely fully specified in planning. To our knowledge, GRT [41] is the only planner to compute the heuristic in the reverse direction, and it does this by first (heuristically) elaborating the partial goal state into a plausible complete and consistent state, and then carrying out the reachability computation from this complete state. One problem with this approach of course is that there can be an exponential number of complete states consistent with the given (partial) goal state, and it is not obvious which of these will be shortest distance from the initial state. The alternative of doing distance computation over all the complete states is of course too expensive. Several planners—including HSP-r and *AltAlt* reverse the direction of search, and thus focus on controlling regression planners, instead of progression planners.

Interestingly, while regression planners can avoid recomputing the distance estimates on each operator application, they do not necessarily dominate progression planners using distance based heuristics. One reason for this is that progression and regression planners exploit the distance heuristics for different purposes. Progression planners already search through the space of consistent and reachable states, and thus need the heuristics only to give their search a goal-directed flavor.[18] They can accomplish this with the aid of even those heuristics that make simplistic independence assumptions. Regression planners, on the other hand have goal-directedness built-in, but need the heuristics to get the reachability and consistency information about their search states. For this, they tend to need heuristics that do a better job of accounting for subgoal interactions (especially the negative interactions). Since heuristics that do a better job of accounting for subgoal interactions are harder to compute than those that just use subgoal independence assumptions, it is not surprising that heuristic progression planners are competitive with (and in some cases better than) heuristic regression planners. Indeed, the planner that performed the best in the 2000 AIPS planning competition, FF, was a progression planner whose heuristic only accounts for positive interactions between subgoals (it should be mentioned that the superior performance of FF is attributed also in large part to its focus on a reduced set of relevant actions, and the use of forced goal orderings and a forced hillclimbing strategy [18]).

Contrary to the impressions created by the early work, the use of distance based heuristics is by no means limited to state space planners. In Section 8 we discussed distance based variable and value ordering heuristics for Graphplan. In [6], Cayrol et al. independently discovered the level based variable ordering heuristic for Graphplan. For value ordering, they use the Graphplan's noops first heuristic. Interestingly, they report that this combinations leads to reasonable improvements in standard Graphplan search (which includes both failing and succeeding levels). In [9], the utility of distance based variable and value ordering heuristics in solving CSP encodings of planning problems is investigated. In our recent work on RePOP [37], we have demonstrated the utility of distance heuristics in partial order planning. *Sapa* [10] is a recent metric temporal planner that uses distance heuristics derived from planning graphs.

As of this writing, heuristic state search planners, including *AltAlt* seem to be outperforming Graphplan, and other disjunctive planners based on CSP or SAT encodings in classical planning domains. This does not necessarily mean that the other planning techniques are slated for obsolescence. Heuristic state search planners are still quite incapable of generating "parallel" plans of the kind that disjunctive planners such as standard Graphplan and GP-CSP [9,17] can generate. There are also several reasons to believe that partial order planners might offer important advantages in planing in metric temporal domains [37,43]. On a larger note, one lesson of this research is that there are bound to be important synergistic interactions among efforts on scaling and generalizing the different planning techniques. Afterall, we have shown that the planning graph, which was originally developed as part of the disjunctive planning techniques, can be of critical importance in deriving effective distance based heuristics!

[18] Progression planners also suffer from higher branching factor, since there are often more applicable actions than relevant ones. The branching factor can however be controlled by techniques such as our *sel-exp* strategy that focus on just the set of actions that appear in the planning graphs. In fact, an action selection strategy of this type has been credited for a significant part of the effectiveness of FF [18].

9.2. Parallels with AI search literature

Extracting effective and admissible heuristics has traditionally been a central problem in the AI search literature [40]. Interestingly, deriving effective heuristics for domain-independent planning started to receive attention only recently, beginning with the work by McDermott [33] and Bonet and Geffner [3,4], who demonstrated that it is possible to derive very good domain-independent heuristics for state space planners. One important difference between the work in AI search and AI planning communities is that the former often focuses on heuristics for specific domains, while the latter considers only domain-independent heuristics. Nevertheless, there are interesting parallels between the work on heuristic generation in these two communities. We explore these relations below.

In the AI search literature, the standard method for computing heuristic functions is to compute the cost of exact solutions to a relaxed 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 the same square. This leads to 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 tile can move independently of each other 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 the 15-puzzle, because the interactions among the tiles are not accounted for.

Most early domain-independent heuristics in AI planning, such as those introduced by Ghallab et al. [14], McDermott [33] and Bonet and Geffner [3,4], make the assumption that subgoals are independent of one another in a manner similar to the Manhattan distance heuristic. These heuristics are neither admissible nor robust. There are many domains in which they cannot solve even the simplest problem instances (cf. Fig. 1).

Obviously, a better heuristic has to go beyond the independence assumption by aggressively taking into account subgoal interactions. In the search literature, Culberson and Schaeffer [7] introduced the notion of *pattern databases* as a method for accounting for such interactions. For instance, in the 15-puzzle problem, they start by focusing on a distinguished subset of tiles, such as the seven tiles in the right column and bottom row. These distinguished subsets are called the *fringe patterns*. The minimum number of moves required to get the fringe pattern 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 is smaller, 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 can take 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 [7] and Rubik's Cube [27].

Our admissible *set-level* heuristic is in a way related to the pattern database idea, where each pair of subgoals can be seen as a "pattern". The difference here is that $lev(\{p_1, p_2\})$ is *not* the exact cost (distance) of achieving the pair $\{p_1, p_2\}$, but a lower bound provided by the planning graph.[19] The *set-level* heuristic thus involves taking the maximum values among all the "patterns" existing in a given set. Furthermore, the accuracy of the *set-level* heuristic can be improved by increasing the size and number of patterns, corresponding to the computation of the higher-order mutexes.

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 sum of the distance values corresponding to each of the disjoint patterns existing 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 sum 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. For most problems, this requirement rarely holds. As noted by Korf [28], disjoint databases cannot be used on the Rubik's Cube. In the AI planning literature, Edelkamp [11] recently developed a method for identifying disjoint patterns, and used that to apply pattern databases technique to devise admissible heuristics in a number of planning domains.

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. Consider, in a 15-puzzle problem, a database that contains the number of moves required to correctly position every 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 subgoal independence). This is the basic idea behind the *linear-conflict* heuristic function, first proposed by Hansson et al. [16]. Korf and Taylor [29] have applied this idea to solving 24-puzzle successfully. 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 tile positions, the set of 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.

The improvement to the Manhattan distance using the linear-conflict idea in the 24-puzzle problems is related to our ideas in the *adjusted-sum* heuristic family, where we add to the *sum* heuristic a quantity $\Delta(S)$, which accounts for the negative interactions among subgoals.

[19] 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.

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 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 [21]. 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 the 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.

While this work mainly focused on deriving heuristics for planning algorithms by state space and CSP search, we believe that the framework for deriving heuristics that we present here is quite general and can be exploited in other contexts. One such example is the work in [37], where we use planning graph based heuristics to develop a significantly fast partial order planner called *RePOP*. *RePOP* is the first partial order planner to rival the performance of Graphplan style planners. Another example is the work in [10], which presents a state of the art heuristic metric temporal planner called *Sapa*. *Sapa* extends the notion of planning graphs to metric temporal domains, and uses these generalized planning graphs to derive heuristics to control search. *Sapa* is one of the first domain independent planners to promise scalable performance in metric temporal domains.

Acknowledgements

We thank Blai Bonet, Minh B. Do, Hector Geffner, Jillian Nottingham, Ioannis Refanidis, David Smith, Biplav Srivastava and Terry Zimmerman 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, AFOSR grant F20602-98-1-0182, NSF grant IRI-9801676, and the NASA grants NAG2-1461 and NCC-1225.

References

[1] F. Bacchus, Results of the AIPS 2000 Planning Competition, 2000. URL: http://www.cs.toronto.edu/aips-2000.
[2] A. Blum, M.L. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1–2) (1997) 281–300.
[3] B. Bonet, H. Geffner, Planning as heuristic search: New results, in: Proc. ECP-99, Durham, UK, 1999.
[4] B. Bonet, G. Loerincs, H. Geffner, A robust and fast action selection mechanism for planning, in: Proc. AAAI-97, Providence, RI, 1997.
[5] B. Bonet, H. Geffner, HSP planner, in: AIPS-98 Planning Competition, Pittsburgh, PA, 1998.
[6] M. Cayrol, P. Regnier, V. Vidal, New results about LCGP, a least committed Graphplan, in: Proc. AIPS-2000, Breckenridge, CO, 2000.
[7] J. Culberson, J. Schaeffer, Pattern databases, Comput. Intelligence 14 (1998) 318–334.
[8] M. Do, S. Kambhampati, B. Srivastava, Investigating the effect of relevance and reachability constraints on SAT encodings of planning, in: Proc. AIPS-2000, Breckenridge, CO, 2000.
[9] M. Do, S. Kambhampati, Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP, Artificial Intelligence 132 (2001) 151–182.
[10] M. Do, S. Kambhampati, SAPA: A domain-independent heuristic metric temporal planner, in: Proc. ECP-2001, Toledo, Spain, 2001.
[11] S. Edelkamp, Planning with pattern databases, in: Proc. ECP-2001, Toledo, Spain, 2001.
[12] R. Fikes, N. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 1 (1971) 27–120.
[13] M. Fox, D. Long, Automatic inference of state invariants in TIM, J. Artificial Intelligence Res. 9 (1998) 367–421.
[14] M. Ghallab, H. Laruelle, Representation and control in IxTeT, in: Proc. AIPS-94, Chicago, IL, 1994.
[15] A. Gerevini, L. Schubert, Inferring state constraints for domain-independent planning, in: Proc. AAAI-98, Madison, WI, 1998.
[16] O. Hansson, A. Mayer, M. Yung, Criticizing solutions to relaxed models yields powerful admissible heuristics, Inform. Sci. 63 (3) (1992) 207–227.
[17] P. Haslum, H. Geffner, Admissible heuristics for optimal planning, in: Proc. AIPS-2000, Breckenridge, CO, 2000.
[18] J. Hoffman, A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm, Technical Report No. 133, Albert Ludwigs University.

[19] S. Kambhampati, Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan, J. Artificial Intelligence Res. 12 (2000) 1–34.
[20] S. Kambhampati, E. Lambrecht, E. Parker, Understanding and extending graphplan, in: Proc. ECP-97, Toulouse, France, 1997.
[21] S. Kambhampati, Challenges in bridging plan synthesis paradigms, in: Proc. IJCAI-97, Nagoya, Japan, 1997.
[22] S. Kambhampati, EBL & DDB for Graphplan, in: Proc. IJCAI-99, Stockholm, Sweden, 1999.
[23] S. Kambhampati, R.S. Nigenda, Distance based goal ordering heuristics for Graphplan, in: Proc. AIPS-2000, Breckenridge, CO, 2000.
[24] H. Kautz, B. Selman, Blackbox: Unifying sat based and graph based planning, in: Proc. IJCAI-99, Stockholm, Sweden, 1999.
[25] J. Koehler, Solving complex planning tasks through extraction of subproblems, in: Proc. 4th AIPS, Pittsburgh, PA, 1998.
[26] R. Korf, Linear-space best-first search, Artificial Intelligence 62 (1993) 41–78.
[27] R. Korf, Finding optimal solutions to Rubik's Cube using pattern databases, in: Proc. AAAI-97, Providence, RI, 1997.
[28] R. Korf, Recent progress in in the design and analysis of admissible heuristic functions (Invited Talk), in: Proc. AAAI-2000, Austin, TX, 2000.
[29] R. Korf, L. Taylor, Finding optimal solutions to the twenty-four puzzle, in: Proc. AAAI-96, Portland, OR, 1996.
[30] D. Long, M. Fox, Efficient implementation of the plan graph in STAN, J. Artificial Intelligence Res. 10 (1999) 87–115.
[31] D. McDermott, A heuristic estimator for means-ends analysis in planning, in: Proc. AIPS-96, Edinburgh, Scotland, 1996.
[32] D. McDermott, AIPS-98 Planning Competition Results, 1998.
[33] D. McDermott, Using regression graphs to control search in planning, Artificial Intelligence 109 (1–2) (1999) 111–160.
[34] S. Mittal, B. Falkenhainer, Dynamic constraint satisfaction problems, in: Proc. AAAI-90, Boston, MA, 1990.
[35] B. Nebel, Y. Dimopoulos, J. Koehler, Ignoring irrelevant facts and operators in plan generation, in: Proc. ECP-97, Toulouse, France, 1997.
[36] X. Nguyen, S. Kambhampati, Extracting effective and admissible state-space heuristics from the planning graph, in: Proc. AAAI-2000, Austin, TX, 2000.
[37] X. Nguyen, S. Kambhampati, Reviving partial order planning, in: Proc. IJCAI-01, Seattle, WA, 2001.
[38] R. Nigenda, X. Nguyen, S. Kambhampati, AltAlt: Combining the advantages of Graphplan and heuristic state search, in: Proc. KBCS-2000, Mumbai, India, 2000.
[39] N. Nilsson, Principles of Artificial Intelligence, Tioga, Los Altos, CA, 1980.
[40] J. Pearl, Heuristics, Morgan Kaufmann, San Mateo, CA, 1984.
[41] I. Refanidis, I. Vlahavas, GRT: A domain independent heuristic for strips worlds based on greedy regression tables, in: Proc. ECP-99, Durham, UK, 1999.
[42] J. Rintanen, An iterative algorithm for synthesizing invariants, in: Proc. AAAI-2000, Austin, TX, 2000.
[43] D. Smith, J. Frank, A. Jonsson, Bridging the gap between planning and scheduling, Knowledge Engineering Review 15 (1) (2000) 47–83.
[44] E. Tsang, Foundations of Constraint Satisfaction, Academic Press, San Diego, CA, 1993.
[45] D. Weld, C. Anderson, D. Smith, Extending graphplan to handle uncertainty & sensing actions, in: Proc. AAAI-98, Madison, WI, 1998.
[46] D. Weld, Recent advances in AI planning, AI Magazine 20 (2) (1999) 93–123.