# Planning as Heuristic Search

Blai Bonet and Héctor Geffner
Depto. de Computación
Universidad Simón Bolívar
Aptdo. 89000, Caracas 1080-A, Venezuela
{bonet,hector}@usb.ve

**Abstract**

In the AIPS98 Planning Contest, the HSP planner showed that heuristic search planners can be competitive with state of the art Graphplan and SAT planners. Heuristic search planners like HSP transform planning problems into problems of heuristic search by automatically extracting heuristics from Strips encodings. They differ from specialized problem solvers such as those developed for the 24-Puzzle and Rubik's cube in that they use a general declarative language for stating problems and a general mechanism for extracting heuristics from these representations.

In this paper, we study a family of heuristic search planners that are based on a simple and general heuristic that assumes that action preconditions are independent. The heuristic is then used in the context of best-first and hill-climbing search algorithms, and is tested over a large collection of domains. We then consider variations and extensions such as reversing the direction of the search for speeding node evaluation, and extracting information about propositional invariants for avoiding dead-ends. We analyze the resulting planners, evaluate their performance, and explain when they do best. We also compare the performance of these planners with two state of the art planners, and show that the simplest planner based on a pure best-first search yields the most solid performance over a large set of problems. We also discuss the strengths and limitations of this approach, establish a correspondence between heuristic search planning and Graphplan, and briefly survey recent ideas that can reduce the current gap in performance between general heuristic-search planners and specialized solvers.

## 1 Introduction

The last few years have seen a number of promising new approaches in Planning. Most prominent among these are Graphplan [BF97] and Satplan [KS96]. Both work in stages by building suitable structures and then searching them for solutions. In Graphplan, the structure is a graph, while in Satplan, it is a set of clauses. Both planners have shown impressive performance and have attracted a good deal of attention. Recent implementations and significant extensions have been reported in [KNHD97, LF99, KS99, ASW98].

In the recent AIPS98 Planning Competition [McD98a], three out of the four planners in the Strips track, IPP [KNHD97], STAN [LF99], and BLACKBOX [KS99], were based on these ideas. The fourth planner, HSP [BG99], was based on the ideas of heuristic search [Nil80, Pea83]. In HSP, the search is assumed to be similar to the search in problems like the 8-Puzzle, the main difference being in the heuristic: while in problems like the 8-Puzzle the heuristic is normally given (e.g., as the sum of Manhattan distances), in planning it has to be extracted automatically from the declarative representation of the problem. HSP thus appeals to a simple scheme for computing the heuristic from Strips encodings and uses the heuristic to guide the search for the goal.

The idea of extracting heuristics from declarative problem representations for guiding the search in planning has been advanced recently by Drew McDermott [McD96, McD99], and by Bonet, Loerincs and Geffner [BLG97]. In this paper, we extend these ideas, test them over a large number of problems and domains, and introduce a family of planners that are competitive with some of the best current planners.

Planners based on the ideas of heuristic search are related to specialized solvers such as those developed for domains like the 24-puzzle [KT96], Rubik's cube [Kor98], and Sokoban [JS99] but differ from them mainly in the use of a general language for stating problems and a general mechanism for extracting heuristics. Heuristic search planners, as all planners, are *general problem solvers* in which the same code must be able to process problems from different domains [NS63]. This generality comes normally at a price: as noted in [JS99], the performance of the best current planners is still well behind the performance of specialized solvers. Closing this gap, however, is the main challenge in planning research where the ultimate goal is to have systems that combine flexibility and efficiency: flexibility for modeling a wide range of problems, and efficiency for obtaining good solutions fast.[1] In heuristic search planning, this challenge can only be met by the formulation, analysis, and evaluation of suitable domain-independent heuristics and optimizations. In this paper we aim to present the basic ideas and results we have obtained, and discuss more recent ideas that we find promising.[2]

More precisely, we will present a family of heuristic search planners based on a simple and general heuristic that assumes that action preconditions are independent. This heuristic is then used in the context of best-first and hill-climbing search algorithms, and is tested over a large class of domains. We also consider variations and extensions, such as reversing the direction of the search for speeding node evaluation, and extracting information about propositional invariants for avoiding dead-ends. We compare the resulting planners with some of the best current planners and show that the simplest planner based on a pure best-first search, yields the most solid performance over a large set of problems. We also discuss the strengths and limitations of this approach, establish a correspondence between heuristic search planning and Graphplan (a recent and popular planning framework [BF97]) and briefly survey recent ideas that can help reduce the performance gap between general heuristic-search planners and specialized solvers.

Our focus is on *non-optimal sequential planning.* This is contrast with the recent emphasis on *optimal parallel planning* following Graphplan and SAT-based planners [KS96]. Algorithms are evaluated in terms of the problems that they solve (given limited time and memory resources), and the quality of the solutions found (measured by the solution time and length). The use of heuristics for *optimal* sequential and parallel planning is considered in [HG00] and is briefly discussed in Sect. 8.

In this paper we review and extend the ideas and results reported in [BG99]. However, rather than focusing on the two specific planners HSP and HSPr, we consider and analyze a broader space of alternatives and perform a more exhaustive empirical evaluation. This more systematic study led us to revise some of the conjectures in [BG99] and to understand better the strengths and limitations involved in the choice of the heuristics, the search algorithms, and the direction of the search.

The rest of the paper is organized as follows. We cover first general state models (Sect. 2) and the state models underlying problems expressed in Strips (Sect. 3). We then present a domain independent

[1] Interestingly, the area of constraint programming has similar goals although it is focused on a different class of problems [HSD92]. Yet, see [BC99] for a recent attempt to apply the ideas of constraint programming in planning.

[2] Another way to reduce the gap between planners and specialized solvers is by making room in planning languages for expressing domain-dependent control knowledge (e.g., [BK98]). In this paper, however, we don't consider this option which is perfectly compatible and complementary with the ideas that we discuss.

heuristic that can be obtained from Strips encodings (Sect. 4), and use this heuristic in the context of forward and backward state planners (Sect. 5 and 6). We then consider related work (Sect. 7), summarize the main ideas and results, and discuss current open problems (Sect. 8).

# 2 State Models

State spaces provide the basic action model for problems involving deterministic actions and complete information. A *state space* consists of a finite set of states $S$, a finite set of actions $A$, a state transition function $f$ that describes how actions map one state into another, and a cost function $c(a, s) > 0$ that measures the cost of doing action $a$ in state $s$ [NS72, Nil80]. A state space extended with a given initial state $s_0$ and a set $S_G$ of goal states will be called a *state model.* State models are thus the models underlying most of the problems considered in heuristic search [Pea83] as well as the problems that fall into the scope of classical planning [Nil80]. Formally, a state model is a tuple $\mathcal{S} = \langle S, s_0, S_G, A, f, c\rangle$ where

- $S$ is a finite and non-empty set of states $s$
- $s_0 \in S$ is the initial state
- $S_G \subseteq S$ is a non-empty set of goal states
- $A(s) \subseteq A$ denotes the actions applicable in each state $s \in S$
- $f(a, s)$ denotes a state transition function for all $s \in S$ and $a \in A(s)$, and
- $c(a, s)$ stands for the cost of doing action $a$ in state $s$.

A *solution* of a state model is a sequence of actions $a_0, a_1, \ldots, a_n$ that generates a state trajectory $s_0$, $s_1 = f(s_0), \ldots, s_{n+1} = f(s_n, a_n)$ such that each action $a_i$ is applicable in $s_i$ and $s_{n+1}$ is a goal state, i.e., $a_i \in A(s_i)$ and $s_{n+1} \in G$. The solution is *optimal* when the total cost $\sum_{i=0}^{n} c(s_i, a_i)$ is minimized.

In problem solving, it is common to build state models adapted to the target domain by explicitly defining the state space and explicitly coding the state transition function $f(a, s)$ and the action applicability conditions $A(s)$ in a suitable programming language. In planning, state models are defined implicitly in a general declarative language that can easily accommodate representations of different problems. We consider next the state models underlying problems expressed in the Strips language [FN71].[3]

# 3 The Strips State Model

A planning problem in Strips is represented by a tuple $P = \langle A, O, I, G\rangle$ where $A$ is a set of atoms, $O$ is a set of operators, and $I \subseteq A$ and $G \subseteq A$ encode the initial and goal situations. The operators $op \in O$ are all assumed grounded (i.e., with the variables replaced by constants). Each operator has a precondition, add, and delete lists denoted as $Prec(op)$, $Add(op)$, and $Del(op)$ respectively. They are all given by sets of atoms from $A$. A Strips problem $P = \langle A, O, I, G\rangle$ defines a state-space $\mathcal{S}_P = \langle S, s_0, S_G, A(\cdot), f, c\rangle$ where

S1. the states $s \in S$ are collections of atoms from $A$

S2. the initial state $s_0$ is $I$

S3. the goal states $s \in S_G$ are such that $G \subseteq s$

S4. the actions $a \in A(s)$ are the operators $op \in O$ such that $Prec(op) \subseteq s$

[3] As it is common, we use the current version of the Strips language as defined by the Strips subset of PDDL [McD98b] rather than original version in [FN71].

S5. the transition function $f$ maps states $s$ into states $s' = s - Del(a) + Add(a)$ for $a \in A(s)$

S6. all action costs $c(a, s)$ are 1

The (optimal) solutions of the problem $P$ are the (optimal) solutions of the state model $\mathcal{S}_P$. A possible way to find such solutions is by performing a search in such space. This approach, however, has not been popular in planning where approaches based on divide-and-conquer ideas and search in the space of plans have been more common [Nil80, Wel94]. This situation however has changed in the last few years, after Graphplan [BF97] and SAT approaches [KS96] achieved orders of magnitude speed ups over previous approaches. More recently, the idea of planning as state-space search has been advanced in [McD96] and [BLG97]. In both cases, the key ingredient is the heuristic used to guide the search that is extracted automatically from the problem representation. Here we follow the formulation in [BLG97].

# 4 Heuristics

The heuristic function $h$ for solving a problem $P$ in [BLG97] is obtained by considering a 'relaxed' problem $P'$ in which all *delete lists* are ignored. From any state $s$, the optimal cost $h'(s)$ for solving the relaxed problem $P'$ can be shown to be a lower bound on the optimal cost $h^*(s)$ for solving the original problem $P$. As a result, the function $h'(s)$ could be used as an admissible heuristic for solving the original problem $P$. However, solving the 'relaxed' problem $P'$ and obtaining the function $h'$ are NP-hard problems.[4] We thus use an approximation and set $h(s)$ to an *estimate* of the optimal value function $h'(s)$ of the relaxed problem. In this approximation, we estimate the cost of achieving the goal atoms from $s$ and then set $h'(s)$ to a suitable combination of those estimates. The cost of individual atoms is computed by a procedure which is similar to the ones used for computing shortest paths in graphs [AMO93]. Indeed, the initial state and the actions can be understood as defining a graph in atom space in which for every action $op$ there is a directed link from the preconditions of $op$ to its positive effects. The cost of achieving an atom $p$ is then reflected in the length of the paths that lead to $p$ from the initial state. This intuition is formalized below.

We will denote the cost of achieving an atom $p$ from the state $s$ as $g_s(p)$. These estimates can be defined recursively as[5]

$$g_s(p) = \begin{cases} 0 & \text{if } p \in s \\ \min_{op \in O(p)}[1 + g_s(Prec(op))] & \text{otherwise} \end{cases} \tag{1}$$

where $O(p)$ stands for the actions $op$ that add $p$, i.e., with $p \in Add(op)$, and $g_s(Prec(op))$, to be defined below, stands for the estimated cost of achieving the preconditions of action $op$ from $s$.

While there are many algorithms for obtaining the function $g_s$ defined by (1), we use a simple forward chaining procedure in which the measures $g_s(p)$ are initialized to 0 if $p \in s$ and to $\infty$ otherwise. Then, every time an operator $op$ is applicable in $s$, each atom $p \in Add(op)$ is added to $s$ and $g_s(p)$ is updated to

$$g_s(p) := \min \left[ \; g_s(p) \; , \; 1 + g_s(Prec(op)) \; \right] \tag{2}$$

These updates continue until the measures $g_s(p)$ do not change. It's simple to show that the resulting measures satisfy Equation 1. The procedure is polynomial in the number of atoms and actions, and corresponds essentially to a version of the Bellman-Ford algorithm for finding shortest paths in graphs [AMO93].

[4] This can be shown by reducing set-covering to Strips with no deletes.

[5] Where the min of an empty set is defined to be infinite.

The expression $g_s(Prec(op))$ in both (1) and (2) stands for the estimated cost of the *set* of atoms given by the preconditions of action *op*. In planners such as HSP, the cost $g_s(C)$ of a set of atoms $C$ is defined in terms of the cost of the atoms in the set. As we will see below, this can be done in different ways. In any case, the resulting heuristic $h(s)$ that estimates the cost of achieving the goal $G$ from a state $s$ is defined as

$$h(s) \stackrel{\text{def}}{=} g_s(G) \tag{3}$$

The cost $g_s(C)$ of *sets* of atoms can be defined as the weighted sum of costs of individual atoms, the minimum of the costs, the maximum of the costs, etc. We consider two ways. The first is as the *sum* of the costs of the individual atoms in $C$:

$$g_s^+(C) = \sum_{r \in C} g_s(r) \quad \text{(additive costs)} \tag{4}$$

We call the heuristic that results from setting the costs $g_s(C)$ to $g_s^+(C)$, the *additive heuristic* and denote it by $h_{add}$. The heuristic $h_{add}$ assumes that subgoals are *independent.* This is not true in general as the achievement of some subgoals can make the achievement of the other subgoals more or less difficult. For this reason, the additive heuristic is not *admissible* (i.e., it may overestimate the true costs). Still, we will see that it is quite useful in planning.

Second, an admissible heuristic can be defined by combining the cost of atoms by the *max* operation as:

$$g_s^{max}(C) = \max_{r \in C} g_s(r) \quad \text{(max costs)} \tag{5}$$

We call the heuristic that results from setting the costs $g_s(C)$ to $g_s^{max}(C)$, the *max heuristic* and denote it by $h_{max}$. The max heuristic unlike the additive heuristic is admissible as the cost of achieving a set of atoms cannot be lower than the cost of achieving each of the atoms in the set. On the other hand, the max heuristic is often less informative. In fact, while the additive heuristic combines the costs of *all* subgoals, the max heuristic focuses only on the most difficult subgoals ignoring all others. In Sect. 7, however, we will see that a refined version of the max heuristic is used in Graphplan.

# 5 Forward State Planning

## 5.1 HSP: A Hill Climbing Planner

The planner HSP [BG99] that was entered into the AIPS98 Planning Contest, uses the additive heuristic $h_{add}$ to guide a hill-climbing search from the initial state to the goal. The hill-climbing search is very simple: at every step, one of the best children is selected for expansion and the same process is repeated until the goal is reached. Ties are broken randomly. The best children of a node are the ones that minimize the heuristic $h_{add}$. Thus, in every step, the estimated atom costs $g_s(p)$ and the heuristic $h(s)$ are computed for the states $s$ that are generated. In HSP, the hill climbing search is extended in several ways; in particular, the number of consecutive plateau moves in which the value of the heuristic $h_{add}$ is not decremented is counted and the search is terminated and restarted when this number exceeds a given threshold. In addition, all states that have been generated are stored in a fast memory (a hash table) so that states that have been already visited are avoided in the search and their heuristic values do not have to be recomputed. Also, a simple scheme for restarting the search from different states is used for avoiding getting trapped into the same plateaus.

Many of the design choices in HSP are ad-hoc. They were motivated by the goal of getting a better performance for the Planning Contest and by our earlier work on a real-time planner based on the

| Round | Planner | Avg. Time | Solved | Fastest | Shortest |
|---|---|---|---|---|---|
| Round 1 | BLACKBOX | 1.49 | 63 | 16 | 55 |
| | HSP | 35.48 | 82 | 19 | 61 |
| | IPP | 7.40 | 63 | 29 | 49 |
| | STAN | 55.41 | 64 | 24 | 47 |
| Round 2 | BLACKBOX | 2.46 | 8 | 3 | 6 |
| | HSP | 25.87 | 9 | 1 | 5 |
| | IPP | 17.37 | 11 | 3 | 8 |
| | STAN | 1.33 | 7 | 5 | 4 |

Table 1: Results of the AIPS98 Contest (Strips track). Columns show the number of problems solved by each planner, the average time over the problems solved (in seconds) and the number of problems in which each planner was fastest or produced shortest plans (from [McD98a]).

same heuristic [BLG97]. HSP did actually well in the Contest where the other participants (in the Strips track) were three state of the art Graphplan and SAT planners: IPP [KNHD97], STAN [LF99] and BLACKBOX [KS99]. Table 1 from [McD98a] shows a summary of the results. In the contest there were two rounds with 140 and 15 problems each, in both cases drawn from several domains. The table shows for each planner in each round, the number of problems solved, the average time taken over the problems that were solved, and the number of problems in which each planner was fastest or produced shortest solutions. IPP, STAN and BLACKBOX, are *optimal* parallel planners that minimize the number of time steps (in which several actions can be performed concurrently) but not the number of actions.

As it can be seen from the table, HSP solved more problems than the other planners but it often took more time or produced longer plans. More details about the setting and results of the competition can be found in [McD98a] and in an article to appear in the AI Magazine.

## 5.2 HSP2: A Best-First Search Planner

The results above and a number of additional experiments suggest that HSP is competitive with the best current planners over many domains. However, HSP is not an optimal planner, and what's worse – considering that optimality has not been a traditional concern in planning – the search algorithm in HSP is not complete. In this section, we show that this last problem can be overcome by switching from hill-climbing to a *best-first search* (BFS) [Pea83]. Moreover, the resulting BFS planner is superior in performance to HSP, and it appears to be superior to some of the best planners over a large class of problems. By performance we mean: the number of problems solved, the time the get those solutions, and the length of those solutions measured by the number of actions.

We will refer to the planner that results from the use of the additive heuristic $h_{add}$ in a best-first search from the initial state to the goal, as HSP2. This best-first search keeps an Open and a Closed list of nodes as A* [Nil80, Pea83] but weights nodes by an evaluation function $f(n) = g(n) + W\,h(n)$, where $g(n)$ is the accumulated cost, $h(n)$ is the estimated cost to the goal, and $W \geq 1$ is a constant. For $W = 1$, the algorithm is A* and for $W \neq 1$ it corresponds to the so-called WA* algorithm [Pea83]. Higher values of $W$ usually lead to the goal faster but with solutions of lower quality [Kor93]. Indeed, if the heuristic is admissible, the solutions found by WA* are guaranteed not to exceed the optimal costs in more than $W$ times. HSP2 uses the WA* algorithm with the non-admissible heuristic $h_{add}$ which is evaluated from scratch in every new state generated. The value of the parameter $W$ is fixed at 5, even though values in the range $[2, 10]$ do not make a significant difference.

## 5.3 Experiments

In the experiments below, we assess the performance of the two heuristic search planners, HSP and HSP2, in comparison with two state of the art planners, STAN 3.0 [LF99], and BLACKBOX 3.6 [KS99]. HSP and HSP2 both perform a forward state-space search guided by the additive heuristic $h_{add}$. The first performs an extended hill-climbing search and the later performs a best-first search with an evaluation function in which the heuristic is multiplied by the constant $W = 5$. STAN and BLACKBOX are both based on Graphplan [BF97], but the latter maps the plan graph into a set of clauses that are checked for satisfiability. The version of HSP used in the experiments, HSP 1.2, improves the version used in the AIPS98 Contest and the one used in [BG99].

For each planner and each planning problem we evaluate whether the planner solves the problem, and if so, the time taken by the planner and the number of actions in the solution. STAN and BLACKBOX are optimal parallel planners that minimize the number of time steps but not necessarily the number of actions. Both planners were run with their default options.

The experiments were performed on an Ultra-5 with 128Mb RAM and 2Mb of Cache running Solaris 7 at 333Mhz. The exception are the results for BLACKBOX on the Logistics problems that were taken from the BLACKBOX distribution as the default options produced much poorer results.[6] The domains considered are: Blocks, Logistics, Gripper, 8-Puzzle, Hanoi, and Tire-world. They constitute a representative sample of difficult but solvable instances for current planners. Five of the 10 block-world instances are of our own [BG99], the rest of the problems are taken from others as noted. A planner is said not to solve a problem when it either runs out of memory or runs out of time (10 mins). Failure to find solutions due either to memory or time constraints are displayed as plans with length $-1$.

All the planners are implemented in C and accept problems in the PDDL language (the standard language used in the AIPS98 Contest; [McD98b]). Moreover, the planners in the HSP family convert every problem instance in PDDL into a program in C. Generating, compiling, linking, and loading such program takes in the order of 2 seconds. This time is roughly constant for all instances and domains, and is not included in the figures below.

### Blocks-World

The first experiments deal with the blocks-world. The blocks-world is challenging for domain-independent planners due to the interactions among subgoals and the size of the state space. The ten instances considered involve from 7 to 19 blocks. Five of these instances are taken from the BLACKBOX distribution and five are from [BG99]. The results for this domain are shown in Fig. 1 that displays for each planner the *length* of the solutions on the left, and the *time* to get those solutions on the right.

The lengths produced by STAN and BLACKBOX are not necessarily optimal in this domain as there is some parallelism (e.g., moving blocks among disjoint pairs of towers). This is the reason the lengths they report do not always coincide. In any case, the solutions reported by the four planners are roughly equivalent over instances 1–5, with STAN and HSP producing slightly longer solutions for instances 4 and 5. Over the more difficult instances 6–10, the situation changes and only HSP and HSP2 report solutions, with the plans found by HSP2 being shorter. Regarding solution times, the times for HSP and HSP2 are roughly even, and slightly shorter than those for STAN and BLACKBOX over the first five instances. Over the last five instances STAN and BLACKBOX run out of memory.

### Logistics

The second set of experiments deals with the logistics domain, a domain that involves the transportation of packages by either trucks or airplanes. Trucks can move among locations in the same city, and airplanes

[6] Those results were obtained on a SPARC Ultra 2 with a 296MHz clock and 256M of RAM [BK98].

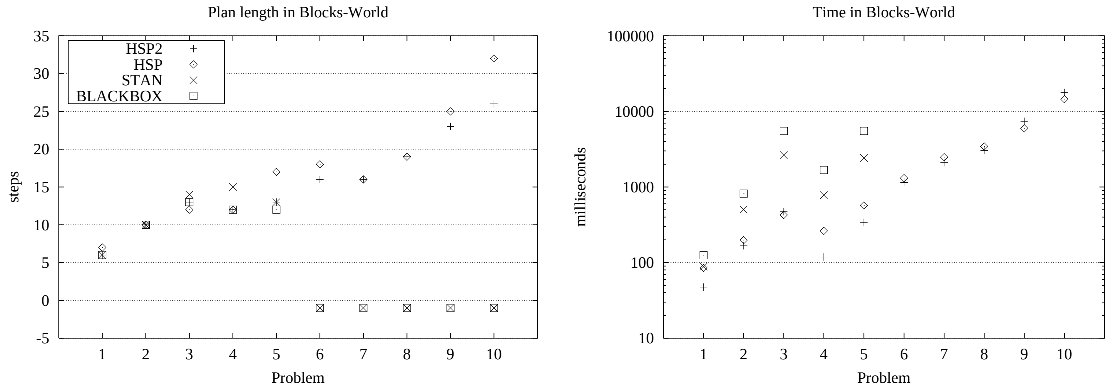


Figure 1: Solution length (left) and time (right) over 10 blocks-world instances.

can move between airports in one city to airports in another city. Packages can be loaded and unloaded in trucks and airplanes, and the task is to transport them from their original locations to some target locations. This is a highly parallel domain, where many operations can be done in parallel. As a result, plans involve many actions but the number of time steps is usually much smaller. The domain is from Kautz and Selman from an earlier version due to Manuela Veloso. The 30 instances we consider are from the BLACKBOX distribution.

The results for this domain are shown in Fig. 2; instances 1–15 at the top and instances 16–30 at the bottom. As before, the number of actions in the plans are reported on the left, times are reported on the right, and failures to find a plan are reported with length $-1$. Both HSP2 and BLACKBOX solve all 30 instances, HSP2 being roughly two orders of magnitude faster. STAN and HSP, on the other, fail on 13 and 10 instances respectively. Interestingly, the times reported by STAN on the instances it solves tend to be close to those reported by HSP2. On the other instances STAN runs out of memory (instances 3,16,17,20,22,28) or time (instances 2,6,7,8,9,15,21).

### Gripper

The third set of experiments deals with the Gripper domain used in the AIPS98 Planning Contest and due to J. Koehler. This is a domain that concerns a robot with two grippers that must transport a set of balls from one room to another. It is very simple for humans to solve but in the Planning Contest proved difficult to most of the planners. Indeed, the domain is not challenging for specialized solvers, but is challenging for certain types of domain-independent planners.

The results over 10 Gripper instances from the AIPS98 Contest are shown in Fig. 3. The planners HSP and HSP2 have no difficulties and compute plans with similar lengths. On the other hand, BLACKBOX solves the first two instances only, and STAN the first four instances. As shown on the left, the time required by both planners grows exponentially and they run out of time over the larger instances. On the other hand, HSP and HSP2 scale up smoothly with HSP2 being slightly faster than HSP.

One of the reasons for the failure of both STAN and BLACKBOX in Gripper is that the heuristic implicitly represented by the plan graph is a very poor estimator in this domain. As a result, Graphplan-based planners, such as STAN and BLACKBOX that perform a form of IDA* search must do many iterations before finding a solution. Actually, the same exponential growth in Gripper occurs also in HSP planners when the heuristic $h_{max}$ is used in place of the additive heuristic. As before, the problem is that the $h_{max}$ heuristic is almost useless in this domain where subgoals are mostly independent. The heuristic

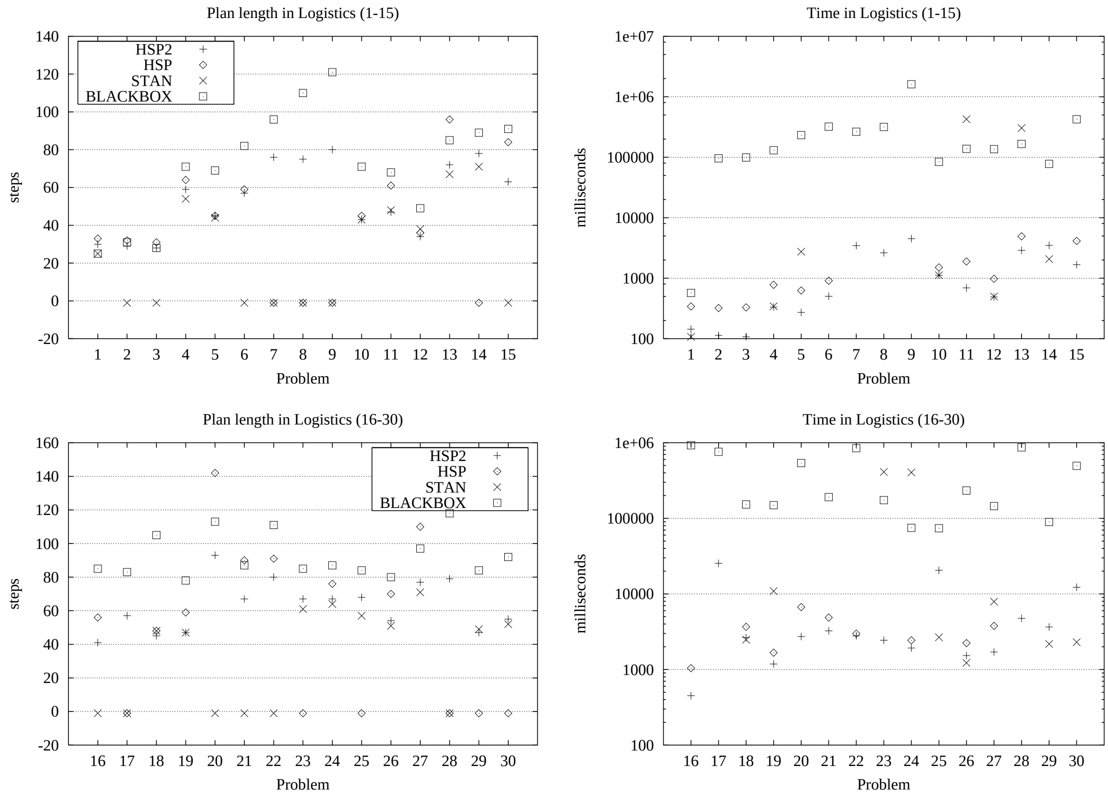


Figure 2: Solution length (left) and time (right) over 30 Logistics instances from Kautz and Selman

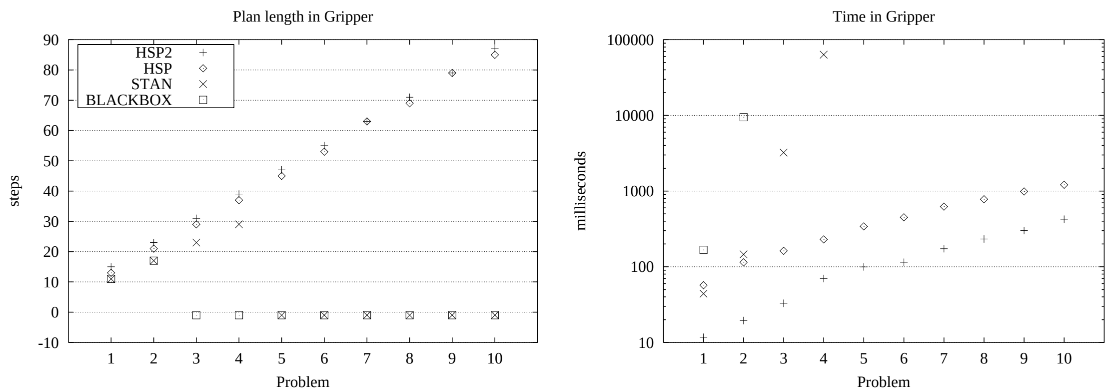


Figure 3: Solution length (left) and time (right) over 10 gripper instances from AIPS98 Contest

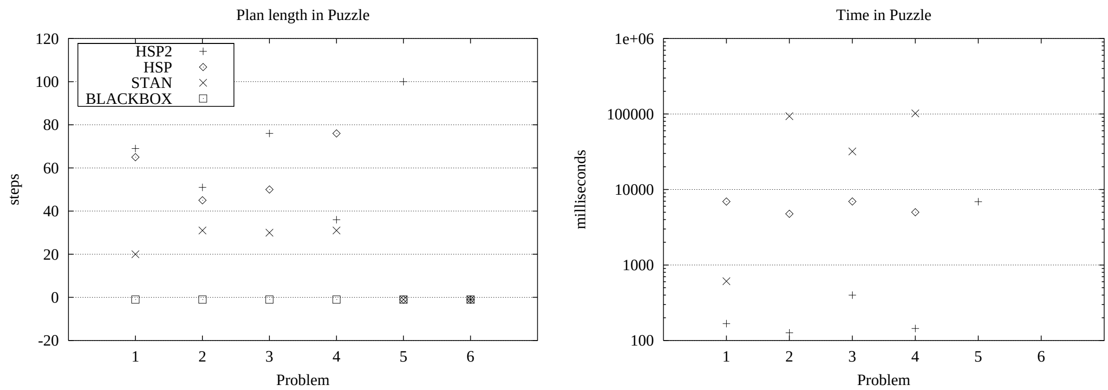


Figure 4: Solution length (left) and time (right) over four instances of the 8-Puzzle (1–4) and two instances of the 15-Puzzle (5–6)

implicit in the plan graph is a refinement of the $h_{max}$ heuristic; the relation between Graphplan and heuristic search planning will be analyzed further in Sect. 7.

### Puzzle

The next problems are four instances of the familiar 8-Puzzle and two instances from the larger 15-Puzzle. Three of the four 8-Puzzle instances are hard as their optimal solutions involves 31 steps, the maximum plan length in such domain. The 15-Puzzle instances are of medium difficulty.

As shown in Fig. 4, HSP and STAN solve the first four instances, and HSP2 solves the first five. The solutions computed by STAN are optimal in this domain which is purely serial. The solutions computed by HSP and HSP2, on the other hand, are poorer, and are often twice as long. On the other hand, as shown on the left part of the figure, HSP2 is two orders of magnitude faster than STAN over the difficult 8-Puzzle instances (2–4) and can also solve instance 5. The times for HSP are worse and does not solve instance 5. BLACKBOX does not solve any of the instances.

### Hanoi

Fig. 5 shows the results for Hanoi. Instance $i$ has $i + 2$ disks, thus problems range from 3 disks up to 8 disks. Over all these problems HSP2 and STAN generate plans of the same quality, HSP2 being slightly faster than STAN. HSP also solves all instances but the solutions are longer. BLACKBOX solves the first two instances.

### Tire-World

The tire-world domain is due to S. Russell and involves operations for fixing flat tires: opening and closing the trunk of a car, fetching and putting away tools, loosening and tightening nuts, etc. Fig. 6 shows the results. Here both STAN and BLACKBOX solve all three instances producing optimal plans. HSP and HSP2 also solve these instances but in some cases they produce inferior solutions. On the time scale, HSP2 is slightly faster than STAN, and both are faster than BLACKBOX in one case by two orders of magnitude. As before, HSP is slower than HSP2 and produces longer solutions.

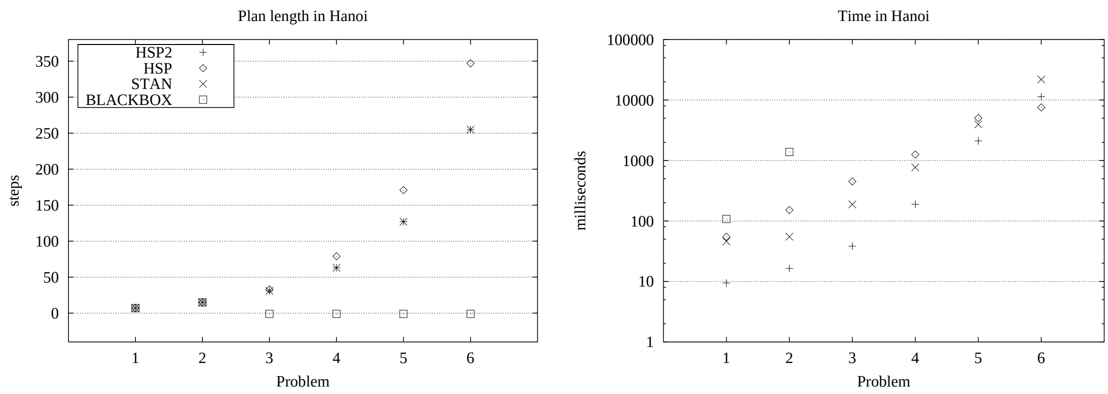


Figure 5: Solution length (left) and time (right) over six Hanoi instances. Instance $i$ has $i+2$ disks.

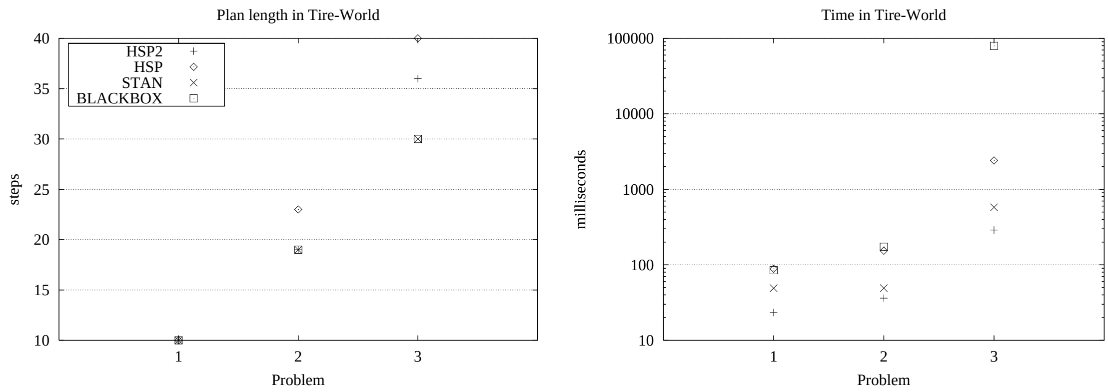


Figure 6: Solution length (left) and time (right) over three instances of Tire-World

### 5.4 Summary: Forward State Planning

The experiments above, based on a representative sample of problems, show that the two forward heuristic search planners HSP and HSP2 are capable of solving the problems solved by two state of the art planners. In addition, in some domains, HSP and in particular HSP2 solve problems that the other planners with their default settings do not currently solve. The planner HSP2, based on a standard best first search, tends to be faster and more robust than the hill-climbing HSP planner. Thus, the arguments in [BG99] in support of a hill-climbing strategy based on the slow node generation rate that results from the computation of the heuristic in every state do not appear to hold in general. Indeed, the combination of the additive heuristic $h_{add}$ and the multiplying constant $W > 1$ often drive the best-first planner to the goal with as few node evaluations as the hill-climbing planner, already providing the necessary 'greedy' bias. An A* search with an admissible and consistent heuristic, on the other hand, is bound to expand all nodes $n$ whose cost $f(n)$ is below the optimal cost. This however does no apply to the WA* strategy used in HSP2.

In the experiments the $W$ parameter in HSP2 was set to the constant value 5. Yet HSP2 is not particularly sensitive to the exact value of this constant. Indeed, in most of the domains, values in the interval $[2, 10]$ produce similar results. This is likely due to the fact that the heuristic $h_{add}$ is not admissible and by itself tends to overestimate the true costs without the need of a multiplying factor. On the other hand, in some domains like Logistics and Gripper, the value $W = 1$ does not lead to solutions. This is precisely because in these domains that involve subgoals that are mostly independent, the additive heuristic is not 'sufficiently' overestimating. Finally, in problems like the sliding tile puzzles, values of $W$ closer to 1 produce better solutions in more time, in correspondence with the normal pattern observed in cases in which the heuristic is admissible [Kor93].

Fig. 7 shows the effects of three different values of $W$ on the quality and times of the solutions, and the number of nodes generated. The values considered are $W = 1$, $W = 2$, and $W = 5$. The top three curves that correspond to Hanoi, are typical for most of the other domains and show little effect. The second set of curves corresponds to Gripper where HSP2 fails to solve the last six instances for $W = 1$. Indeed, the two right most curves show an exponential growth in time and the number of generated nodes. In Logistics, HSP2 with $W = 1$ also fails to solve most of the instances. As noted above, these are two domains where subgoals are mostly independent and where the additive heuristic is not sufficiently overestimating and hence fails to provide the 'greedy bias' necessary to find the solutions. Indeed, the state space in Logistics is very large, while in Gripper it's the branching factor that is large due to the (undetected) symmetries in the problem.

The bottom set of curves in Fig. 7 correspond to the Puzzle domain. In this domain, HSP2 with $W = 1$ and $W = 2$ produce better solutions and in some cases take more time. This probably happens in Puzzle because, as in Gripper and Logistics, there is a degree of decomposability in the domain (that's why the sum of the Manhattan distance works), that makes the additive heuristic behave as an admissible heuristic in WA*. Unlike Gripper and Logistic, however the branching factor of the problem and the size of the state space allow the resulting BFS algorithm to solve the instances even with $W = 1$. Actually, with $W = 1$ and $W = 2$, HSP2 solves the sixth instance of Puzzle which is not solved with $W = 5$.

## 6 HSPr: Heuristic Regression Planning

A main bottleneck in both HSP and HSP2 is the computation of the heuristic from scratch in every new state.[7] This takes more than 80% of the total time in both planners and makes the node generation rate very low. Indeed, in a problem like the 15-Puzzle, both planners generate less than a thousand nodes per

[7] The same applies also to McDermott's UNPOP.

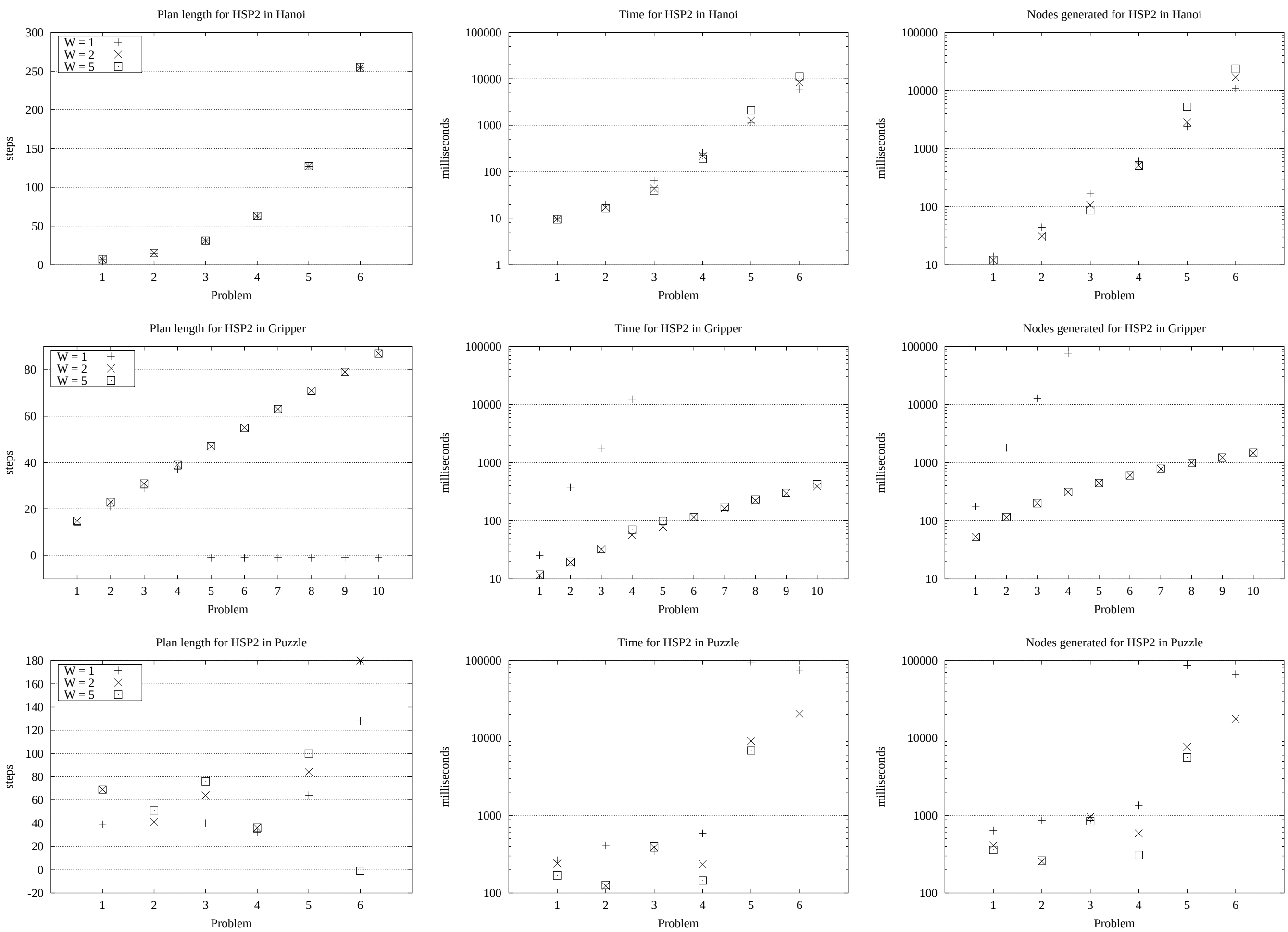


Figure 7: Influence of value of $W$ in the HSP2 planner on the length of the solutions (left), the time required to find solutions (center), and number of nodes generated (right). The domains from top to bottom are Hanoi, Gripper, and Puzzle.

second, while a specialized solver such as [KT96] generates several hundred thousand nodes per second for the more complex 24-puzzle. The reason for the low node generation rate is the computation of the heuristic in which the estimated costs $g_s(p)$ for all atoms $p$ are computed from scratch in every new state $s$.

In [BG99], we noted that this problem could by solved by performing the search *backward* from the goal rather than *forward* from the initial state. In that case, the estimated costs $g_{s_0}(p)$ derived for all atoms from the initial state could be used *without recomputation* for defining the heuristic of any state $s$ arising in the backward search. Indeed, the estimated distance from $s$ to $s_0$ is equal to the distance from $s_0$ to $s$, and this distance can be estimated simply as the sum (or max) of the costs $g_{s_0}(p)$ for the atoms $p$ in $s$. This trick for simplifying the computation of the heuristic and speeding up node generation results from computing the estimated atom costs from a state $s_0$ which then becomes the target of the search. An alternative is to estimate the atom costs from the goal and then perform a forward search toward the goal. This is actually the idea in [RV99]. The problem with this latter scheme is that the goal in planning is not a state but a *set of states;* namely, the states where the goal atoms hold. And computing the heuristic from a set of states in a principled manner is bound to be more difficult than computing the heuristic from a given state (thus the need to 'complete' the goal description in [RV99]).

We thus present below a scheme for performing planning as heuristic search that avoids the recomputation of the atom costs in every new state by computing these costs *once* from the initial state $s_0$. These costs are then used *without recomputation* to define an heuristic that is used to guide a *regression* search from the goal. The benefit of the search scheme is that node generation will be 6-7 times faster. This will show in the solution of some of the problems considered above such as Logistics and Gripper. However, as we will also see, in many problems the new search scheme does not help, and in several cases, it actually hurts. We discuss such issues below.

## 6.1 Regression State Space

We refer to the planner that searches backward from the goal rather than forward from the initial state as HSPr. Backward search is an old idea in planning that is known as *regression search* [Nil80, Wel94]. In regression search, the states can be thought as *sets of subgoals;* i.e., the 'application' of an action in a goal yields a situation in which the execution of the action achieves the goal. Moreover, while a set of atoms $\{p, q, r\}$ in the forward search represents the *unique* state in which the atoms $p$, $q$, and $r$ are true and all other atoms are false, the same set of atoms in the regression search represents the *collection* of states in which the atoms $p$, $q$, and $r$ are true. In particular, the set of goals atoms $G$, which determines the root node of the regression search, stands for the collection of goal states, that is, the states $s$ such that $G \subseteq s$.

For making precise the nature of the backward search, we will thus define explicitly the state space being searched. We will call it the *regression space* and define it in analogy to the *progression space* $\mathcal{S}_P$ defined by [S1]–[S5] above. The regression space $\mathcal{R}_P$ associated with a Strips problem $P = \langle A, O, I, G\rangle$ is given by the tuple $\mathcal{R}_P = \langle \mathsf{S}, \mathsf{s}_0, \mathsf{S}_G, \mathsf{A}(\cdot), \mathsf{f}, \mathsf{c}\rangle$ where

R1. the states $\mathsf{s}$ are sets of atoms from $A$

R2. the initial state $\mathsf{s}_0$ is the goal $G$

R3. the goal states $\mathsf{s} \in \mathsf{S}_G$ are the states for which $\mathsf{s} \subseteq I$

R4. the set of actions $\mathsf{A}(\mathsf{s})$ applicable in $\mathsf{s}$ are the operators $op \in O$ that are *relevant* and *consistent;* namely, for which $Add(op) \cap \mathsf{s} \neq \emptyset$ and $Del(op) \cap \mathsf{s} = \emptyset$

R5. the state $\mathsf{s}' = \mathsf{f}(a, \mathsf{s})$ that follows the application of $a \in \mathsf{A}(\mathsf{s})$ is such that $\mathsf{s}' = \mathsf{s} - Add(a) + Prec(a)$

R6. the action costs $c(a, \mathsf{s})$ are all 1

The solution of this state space is, like the solution of any state model $\langle \mathsf{S}, \mathsf{s}_0, \mathsf{S}_G, \mathsf{A}(\cdot), \mathsf{f}, \mathsf{c}\rangle$, a finite sequence of actions $a_0$, $a_1$, ..., $a_n$ such that for a sequence of states $\mathsf{s}_0$, $\mathsf{s}_1$, ..., $\mathsf{s}_{n+1}$, $\mathsf{s}_{i+1} = \mathsf{f}(a_i, \mathsf{s}_i)$, for $i = 0, \ldots, n$, $a_i \in \mathsf{A}(\mathsf{s}_i)$, and $\mathsf{s}_{n+1} \in \mathsf{S}_G$. The solution of the progression and regression spaces are related in the obvious way; one is the inverse of the other.

We use different fonts for referring to states $s$ in the progression space $\mathcal{S}_P$ and states $\mathsf{s}$ in the regression space $\mathcal{R}_P$. While they are both represented by sets of atoms, they have a different meaning. As we said above, the state $\mathsf{s} = \{p, q, r\}$ in the regression space stands for the *set* of states $s$, $\{p, q, r\} \subseteq s$ in the progression space. For this reason, forward and backward search in planning are *not* symmetric, unlike forward and backward search in problems like the 15-Puzzle or Rubik's Cube.

## 6.2 Heuristic

The planner HSPr searches the regression space [R1]–[R5] using an heuristic based on the additive cost estimates $g_s(p)$ described in Sect 4. These estimates are computed only once from the initial state $s_0 \in \mathcal{S}$. The heuristic $h_{add}(\mathsf{s})$ associated with *any* state $\mathsf{s}$ is then defined as

$$h_{add}(\mathsf{s}) = \sum_{p \in \mathsf{s}} g_{s_0}(p) \tag{6}$$

While in HSP, the heuristic $h_{add}(s)$ combines the cost estimates $g_s(p)$ of a fixed set of goal atoms computed from each state $s$, in HSPr, the heuristic $h_{add}(\mathsf{s})$ combines the cost estimates of the set of subgoals $p$ in $\mathsf{s}$ from a fixed state $s_0$. The heuristic $h_{max}(\mathsf{s})$ can be defined in an analogous way by replacing sums by maximizations.

## 6.3 Mutexes

The regression search often leads to states $\mathsf{s}$ that are not reachable from the initial state $s_0$. For example, in the blocks world, the regression of the state

$$\mathsf{s} = \{on(c, d), on(a, b)\}$$

through the action $move(a, d, b)$ leads to the state

$$\mathsf{s}' = \{on(c, d), on(a, d), clear(b), clear(a)\}$$

This state represents a situation in which two blocks, $c$ and $a$ are on the same block $d$. It is simple to show that such situations are unreachable in the block-worlds given a 'normal' initial state. Such unreachable situations are common in regression planning, and if undetected, cause a lot of useless search. A good heuristic would assign an infinite cost to such situations but our heuristics are not as good. Indeed, the basic assumption underlying both the additive and the max heuristics — that the estimated cost of a set of atoms is a function of the estimated cost of the atoms in the set — is violated in such situations. Indeed, while the cost of each of the atoms $on(c, d)$ and $on(a, d)$ is finite, the cost of the *pair* of atoms $\{on(c, d), on(a, d)\}$ is infinite. Better heuristics that do not make this assumption and correctly reflect the cost of such pairs of atoms have been recently described in [HG00]. Here we follow [BG99] and develop a simple mechanism for detecting some *pairs* of atoms $\{p, q\}$ such that any state containing those pairs can be proven to be unreachable from the initial state, and thus can be given an infinite heuristic value and pruned. The idea is adapted from a similar idea used in Graphplan [BF97] and thus we call such pairs of unreachable atoms mutually exclusive pairs or *mutex* pairs. As in Graphplan, the definition below is not guaranteed to identify all mutex pairs, and furthermore, it says nothing about larger sets of atoms that are *not* achievable from $s_0$ but whose proper subsets are.

A tentative definition is to identify a pair of atoms $R$ as a mutex when $R$ is not true in the initial state $s_0$ and every action that asserts an atom in $R$ deletes the other. This definition is sound (it only recognizes pairs of atoms that are not achievable jointly) but is too weak. In particular, it does not recognize a set of atoms like $\{on(a,b), on(a,c)\}$ as a mutex, since actions like $move(a,d,b)$ add the first atom but do not delete the second.

We thus use a different definition in which a pair of atoms $R$ is recognized as mutex when the actions that add one of the atoms in $R$ and do not delete the other atom, can guarantee through their preconditions that such atom will not be true after the action. To formalize this, we consider *sets* of mutexes rather that individual pairs.

**Definition 1** *A set $M$ of atom pairs is a* mutex set *given a set of operators $O$ and an initial state $s_0$ iff for all atoms pairs $R = \{p, q\}$ in $M$*

1. *$R$ is not true in $s_0$,*
2. *for every $op \in O$ that adds $p$, either $op$ deletes $q$, or $op$ does no add $q$ and for some precondition $r$ of $op$, $R' = \{r, q\}$ is a pair in $M$.*

It is simple to verify that if a pair of atoms $R$ belongs to a mutex set, then the atoms in $R$ are really mutually exclusive, i.e., not achievable from the initial state given the available operators. Also if $M_1$ and $M_2$ are two mutex sets, $M_1 \cup M_2$ will be a mutex set as well, and hence according to this definition, there is a single *largest* mutex set. Rather than computing this set, however, that is difficult, we compute an approximation as follows.

We say that a pair $R'$ is a 'bad pair' in $M$ when $R'$ does not comply with one of the conditions 1–2 above. The procedure for constructing a mutex set starts with a set of pairs $M := M_0$ and iteratively removes all bad pairs from $M$ until no bad pair remains. The initial set $M_0$ of 'potential' mutexes can be chosen in a number of ways. In all cases, the result of this procedure is a mutex set $M$ such that $M \subseteq M_0$. One possibility is to set $M_0$ to the set of all pairs of atoms. In [BG99], to avoid the overhead involved in dealing with the $N^2/2$ pairs of atoms and many useless mutexes, we chose a smaller set $M_0$ of potential mutexes that turns out to be adequate for many domains. Such set $M_0$ was defined as the union of the sets $M_A$ and $M_B$ where

- $M_A$ is the set of pairs $P = \{p, q\}$ such that some action adds $p$ and deletes $q$,
- $M_B$ is the set of pairs $P = \{r, q\}$ such that for some pair $P' = \{p, q\}$ in $M_A$ and some action $a$, $r \in Prec(a)$ and $p \in Add(a)$

The structure of this definition mirrors the structure of the definition of mutex sets.

A mutex in HSPr refers to a pair in the set $M^*$ obtained from the set $M_0 = M_A + M_B$ by sequentially removing all 'bad' pairs. Like the analogous definition in Graphplan, the set $M^*$ does not capture all actual mutexes, yet it can be computed fast, and in many of the domains we have considered appears to prune the obvious unreachable states. A difference with Graphplan is that this definition identifies *structural mutexes* while Graphplan identifies *time-dependent* mutexes. On the other hand, because of the fixed point construction, this definition can identify mutexes that Graphplan cannot. For example, in the complete TSP domain [LF99], pairs like $\langle at(city_1), at(city_2)\rangle$ would be recognized as a mutex by this definition but not by Graphplan, as the actions of going to different cities are not mutually exclusive for Graphplan.

## 6.4 Algorithm

The planner HSPr uses the additive heuristic $h_{add}$ and the mutex set $M^*$ to guide a regression search from the goal. The additive heuristic is obtained from the estimated costs $g_{s_0}(p)$ computed once for all

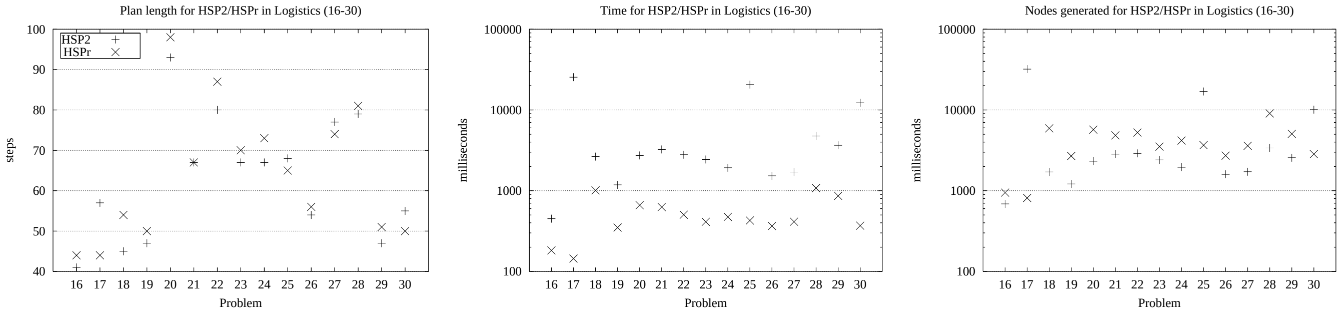


Figure 8: Comparison between HSPr vs HSP2 over Logistics instances 16–30. Curves show solution length (left), time (center), and number of nodes generated (right).

atoms $p$ from the initial state $s_0$. The mutex set $M^*$ is used to 'patch' the heuristic: states $\mathsf{s}$ arising in the search that contain a pair in $M^*$ get an infinite cost and are pruned. The algorithm used for searching the regression space is the same as the ones used in HSP2: a WA* algorithm with the constant $W$ set to 5. Here we depart from the description of HSPr in [BG99] where the WA* algorithm was given a 'greedy' bias. As above, we stick to a pure BFS algorithm. The set of experiments below cover more domains than those in [BG99] and will help us to assess better the strengths and limitations of regression heuristic planning in relation to forward heuristic planning.

## 6.5 Experiments

In the experiments, we compare the regression planner HSPr with the forward planner HSP2. Both are based on a WA* search and both use the same additive heuristic (in the case of HSPr, patched with the mutex information). HSPr avoids the recomputation of the atom costs in every state, and thus computes the heuristic faster and can explore more nodes in the same time. As we will see, this helps in some domains. However, in other domains, HSPr is not more powerful than HSP2, and in some domains HSPr is actually weaker. This is due to two reasons: first, the additional information obtained by the recomputation of the atom costs in every state sometimes pays off, and second, the regression search often generates spurious states that are not recognized as such by the mutex mechanism and cause of lot of useless search. These problems are not significant in the two domains considered in [BG99] but are significant in other domains.

### Logistics

Fig. 8 shows the results of the two planners HSPr and HSP2 over the Logistics instances 16–30. The curves show the length of the solutions (left), the time required to find the solutions (center), and the number of generated nodes (right). It is interesting to see that HSPr generates more nodes than HSP2 and yet it takes roughly four times less time than HSP2 to solve the problems. This follows from the faster evaluation of the heuristic. On the other hand, the plans found by HSPr are often longer than those found by HSP2. Similar results obtain for the logistics instances that are not shown in the figure.

### Gripper

As shown in Fig. 9, a similar pattern arises in Gripper. Here HSPr generates slightly less nodes than HSP2, but since it generates nodes faster, the time gap between the two planner gets larger as the size

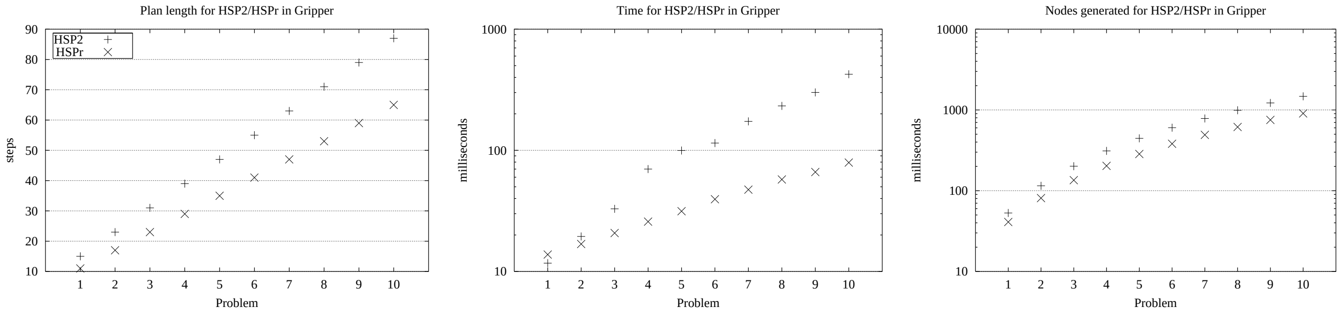


Figure 9: Comparison between HSPr vs HSP2 over Gripper instances. Curves show solution length (left), solution (center), and number of nodes generated (right)

of the problems grows. In this case, the solutions found by HSPr are uniformly better than the solutions found by HSP2, and this difference grows with the size of the problems.

HSPr is also stronger than HSP2 in Puzzle where, unlike HSP2 (with $W = 5$) solves the last instance in the set (a 15-Puzzle instance). However, for the other three domains HSPr does not improve on HSP2, and indeed, in two of these domains (Hanoi and Tire-World) it does significatively worse.

**Hanoi and Tire-World**

The results for Hanoi are shown in Fig. 10. HSPr solves the first three instances (up to 5 disks), but it does not solve the other three. Indeed, as it can be seen, in the first three instances the time to find the solutions and the number of nodes generated grow much faster in HSPr than in HSP2. The same situation arises in the Tire-World where HSP2 solves all three instances and HSPr solves only the first one. The problems, as we mentioned above, are two: spurious states generated in the regression search that are not detected by the mutex mechanisms, and the lack of the 'feedback' provided by the recomputation of the atom costs in every state. Indeed, errors in the estimated costs of atoms in HSP2 can be corrected when they are recomputed; in HSPr, on the other hand, they are never recomputed. So the recomputation of these costs has two effects, one that is bad (time overhead) and one that is good (additional information). In domains where subgoals interact in complex ways, the idea of a forward search in which atom costs are recomputed in every state as implemented in HSP2 will probably make sense; on the other hand, in domains where the additive heuristic is adequate, the backward search with no recomputations as implemented in HSPr can be more efficient.

The results for the HSPr and HSP2 planners in the Tire-World show the same pattern as Hanoi. Indeed, HSPr solves just the first instance, while HSP2 solves the three instances. As we show below, however, part of the problem in this domain has to do with the spurious states generated in the regression search.

## 6.6 Improved Mutex Computation

The regression search often leads to states $s$ that are not reachable from the initial situation and which often violate implicit domain constraints. For example, the regression of a state $s = \{on(c,d), on(a,b)\}$ through the action $move(a,d,b)$ leads to the state $s' = \{on(c,d), on(a,d), clear(b), clear(a)\}$ in which two blocks $c$ and $a$ are on top of the same block $d$. HSPr uses a procedure inspired in Graphplan to detect such mutex pairs and prune spurious states like $s'$ that contain them. Neither our mechanism, nor Graphplan's, however, can detect all spurious (unreachable) states. Indeed, there are unreachable states that do not contain any unreachable pair of atoms such as the state where $a$ is on $b$, $b$ is on $c$,

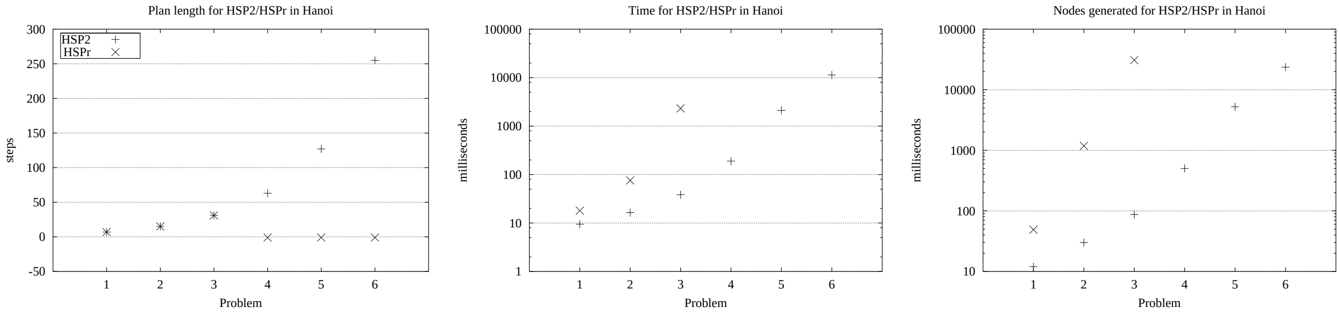


Figure 10: Comparison between HSPr vs HSP2 over Hanoi. Curves show solution length (left), solution (center), and number of nodes generated (right). Instance $i$ has $i+2$ disks.

and $c$ is on $a$ [BF97]. So a potential serious problem in regression planners such as Graphplan and HSPr is the presence of such states. HSPr has an additional problem and that is that it fails to detect some structural mutexes that Graphplan detects.

The procedure used in HSPr to identify mutexes starts with a set $M_0$ of potential mutexes and then removes the 'bad' pairs from $M_0$ until no 'bad' pair remains. A problem we have detected with the definition in [BG99] which we have used here is that the set of potential mutexes $M_0$ sometimes is not large enough and hence useful mutexes are lost. Indeed, we performed experiments in which $M_0$ is set to the collection of *all* atom pairs, and the same procedure is applied to this set until no 'bad' pairs remains. In most of the domains, this change didn't yield a different behavior. However, there were two exceptions. While HSPr solved only the first instance of the Tire-World, HSPr using the extended set of potential mutexes solved the three instances. This shows that in this case HSPr was affected by the problem of spurious states. On the other hand, in problems like logistics, the new set $M_0$ leads to a much larger set of mutexes $M^*$ that are not as useful and yet have to be checked in all the states generated. This slows down node generation with no compensating gain thus making HSPr several times slower. The corresponding curves are shown in Fig. 11, where 'mutex-1' and 'mutex-2' correspond to the original and extended definition of the set $M_0$ of potential mutexes. Since, the benefits appear to be more important than the loses, the extended definition seems worthwhile and we will make it the default option in the next version of HSPr. However, since for the reasons above, the new mechanism is not complete either, the problem caused by the presence of spurious states in regression planning remains open.[8]

## 6.7 Additional Issues in Heuristic Regression Planning

**Additive vs. Max Heuristic.** While we defined two heuristics, the additive heuristic $h_{add}$ and the max heuristic $h_{max}$, we have used only the additive heuristic. The intuition underlying this choice is that the additive heuristic is more informed as it takes into account all subgoals, while the max heuristic only focuses on the subgoals that are perceived as most costly. In order to test this intuition we ran HSPr over all the domains with the additive heuristic and the max heuristic. In problems that involve many independent subgoals such as Gripper and Logistics, the max heuristic is almost useless and very few instances are solved. On the other hand, in problem that involve more complex interactions among goals such as Hanoi and Tire-World, the heuristic $h_{max}$ does slightly better than $h_{add}$, and indeed, in

[8] As noted in [BF97], the problem of detecting *all* mutexes, and even only all mutex *pairs,* is as hard the plan existence problem.

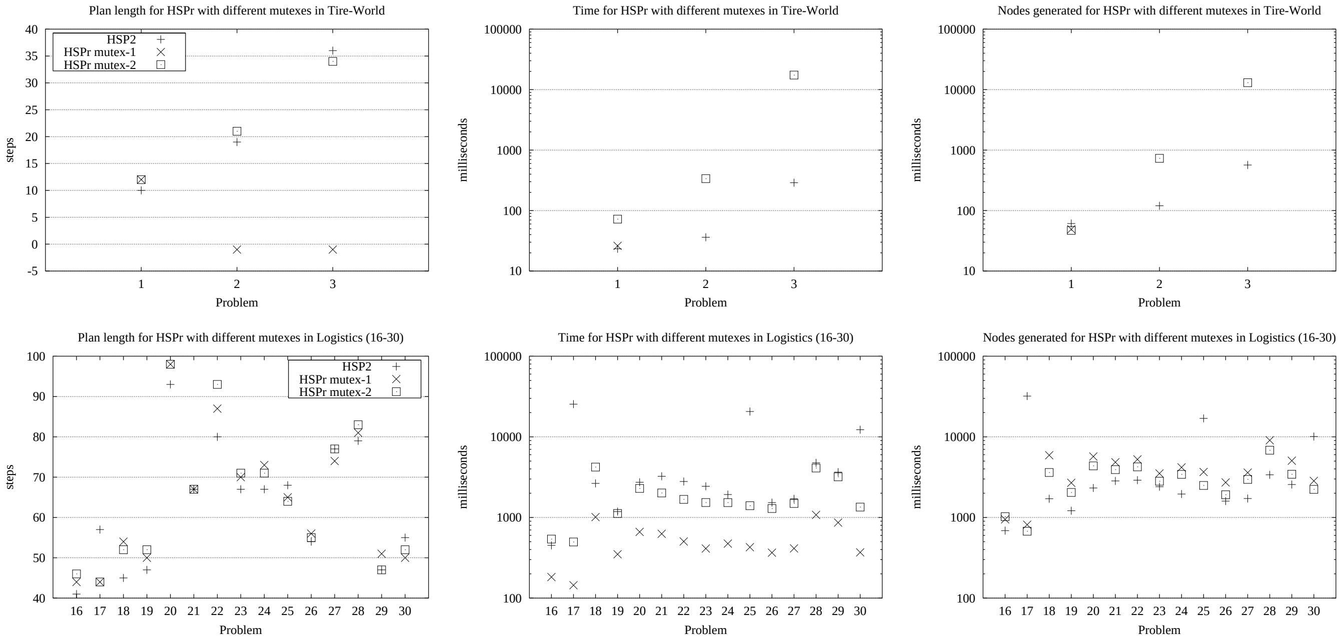


Figure 11: Impact of original vs. extended definition of the set $M_0$ of potential mutexes in Tire-World and Logistics

Tire-World it solves the second instance that $h_{add}$ does not solve (within HSPr). Finally in Blocks-World and Puzzle where there is a certain degree of decomposability, the $h_{max}$ heuristic is worse than the $h_{add}$ heuristic but still manages to solve roughly the same set of instances taking more time.

In summary, the additive heuristic yields a better behavior in HSPr than the max heuristic, but this does not mean that the max heuristic is useless. The heuristic used implicitly in Graphplan is as a refinement of the $h_{max}$ heuristic, as is the family of higher-order heuristic formulated in [HG00]. We will say more about those heuristics below.

**Greedy Best-First Search.** The algorithm used in the version of the HSPr planner presented in [BG99] uses the same WA$^*$ algorithm but with the following variation: when some of the children of the last expanded node improve the heuristic value of the parent, the best such child is selected for expansion even if such node is not a least cost node in the Open list. The idea is to provide an additional greedy bias in the search. This modification helps in some instances and in general does not appear to hurt, yet the boost in performance across a large set of domains is small. For this reason, we have dropped this feature from HSPr which is now a pure BFS regression planner.

**Branching Factor.** A common argument for performing regression search rather than forward search in planning has been based on considerations related to the branching factors of the two spaces [Nil80, Wel94]. We have measured the forward and backward branching factor in all the domains and found that they vary a lot from instance to instance. For example, the forward branching factor in the Blocks-World instances ranges from 16.83 to 84.62, while the backward branching factor ranges from 4.73 to 12.15. In Logistics, the forward branching factor ranges from 7.89 to 37.91, while the backward factor ranges from 9.68 to 25.80. In problems like Puzzle and Hanoi, the average branching factors are roughly constant over the different instances, and are similar in both directions.

We have found, however, that the performance of the two planners, HSP2 and HSPr, on the same problem is not in direct correspondence with the size of the forward and backward branching factors. For example, while for each blocks-world instance the average branching factor in HSPr is less than half the one in HSP2, and moreover the first planner generates nodes 6-7 times faster on average than the second, HSPr is not better than HSP2 in blocks-world. On the other hand, in logistics, where the average branching factor in HSP2 is often smaller than the one in HSPr, HSPr does better. Thus while considerations related to the branching factor of the forward and backward spaces are relevant to the performance of planners, they are not the only or most important consideration. As we mentioned, two considerations that are relevant for explaining the performance of HSPr in relation to HSP2 are the quality of the heuristic (which in HSP2 is recomputed in every state), and the presence of spurious states in the regression search (that do not arise in the forward search). This last problem, however, could be solved by the formulation of better planning heuristics in which the cost of a set of atoms is *not* defined in terms of the costs of the individual atoms in the set as in the $h_{add}$ and $h_{max}$ heuristics. Such heuristics are considered in [HG00] and are briefly discussed below.

# 7 Related Work

## 7.1 Heuristic Search Planning

The idea of extracting heuristics from declarative problem representations in planning has been proposed recently by McDermott [McD96] and by Bonet, Loerincs, and Geffner [BLG97]. In [BLG97], the heuristic is used to guide a real-time planner based on the LRTA* algorithm [Kor90], while in [McD96], the heuristic is used to guide a limited discrepancy search [HG95]. The heuristics in both cases are similar, even though the formulation and the algorithms used for computing them are different. The performance of McDermott's UNPOP, however, does not appear to be competitive with the type of planners discussed in this paper. This may be due to the fact that it is written in Lisp and deals with variables and matching operations at run-time. Most current planners, including those reported in this paper, are written in C and deal with grounded operators only. On the other hand, while most of these planners are restricted to small variations of the Strips language, UNPOP deals with the more expressive ADL [Ped89].

The idea of performing a regression search from the goal for avoiding the recomputation of the atom costs was presented in [BG99] where the HSPr planner was introduced. The version of HSPr considered here, unlike the version reported in [BG99], is based on a pure BFS algorithm. Likewise, the pure BFS forward planner that we have called HSP2, hasn't been discussed elsewhere. HSP2 is the simplest, and as the experiments have illustrated, it is also the most solid planner in the HSP family.

Two advantages of forward planners over regression planners is that the former do not generate spurious states and they often benefit from the additional information obtained by the recomputation of the atom costs in every state. Mutex mechanisms such as those used by HSPr and Graphplan can prune some of spurious states in some problems, but they cannot be complete.

The idea of combining a forward propagation from $s_0$ to compute all atom costs and a backward search from the goal for avoding the recomputation of these costs appears in reverse form in [RV99]. Refanidis and Vlahavas compute cost estimates by a backward propagation from the goal and then use those estimates to perform a forward state-space search from the initial state. In addition, they compute the heuristic in a different way so they get more accurate estimates.

## 7.2 Derivation of Heuristics

The non-admissible heuristic $h_{add}$ used in HSP is derived as an *approximation* of the optimal cost function of a *relaxed* problem where deletes lists are ignored. This formulation has two obvious problems. First, the *approximation* is not very good as it ignores the *positive* interactions among subgoals that can make one goal simpler after a second one has been achieved (this results in the heuristic being non-admissible). Second, the *relaxation* is not good as it ignores the *negative* interactions among subgoals that are lost when delete lists are discarded. These two problems are addressed in the heuristic proposed by Refanidis and Vlahavas [RV99] but their heuristic is still non-admissible and does not have clear justification.

A different approach for addressing these limitations has been reported recently in [HG00]. While the idea of the $h_{max}$ heuristic presented in Sect. 4 is to approximate the cost of a set of atoms by the cost of the most costly atom in the set, the idea in [HG00] is to approximate the cost of a set of atoms by the cost of the the most costly *atom pair* in the set. The resulting heuristic, called $h^2$ is admissible and more informative than the $h_{max}$ heuristic, and can be computed reasonably fast. Indeed, in [HG00] the $h^2$ heuristic is used in the context of an IDA* search to compute *optimal* plans. Higher order heuristics $h^m$ in which the cost of sets of atoms is approximated by the most costly subset of size $m$ are also discused. Such higher-order heuristics may prove useful in problems in which subgoals interact in complex ways.

The derivation of admissible heuristics by the consideration of relaxed models has a long history in AI. Indeed, the Manhattan distance heuristic in sliding tile puzzles is normally explained in terms of the solution of a relaxed problem in which tiles can move to any neighboring position [Pea83]. A similar relaxation is used to explain the Minimum Spanning Tree heuristic used for solving the Traveling Salesman Problem. Moreover, in [Pea83], these relaxations are shown to follow from simplifications in suitable Strips encodings, and in particular the Manhattan heuristic is derived by ignoring some action preconditions.

The idea of deriving heuristics from suitable relaxations is a powerful idea. However, it is often too general to provide practical guidance in the formulation of concrete heuristics for specific problems. Indeed, the idea of dropping action preconditions from Strips encodings is guaranteed to lead to admissible heuristics but computing such heuristics can be as hard as solving the original problem. Indeed, unless we remove all preconditions the class of 'relaxed' planning problems is still intractable. In this paper, we have used a different relaxation in which delete lists are removed. While, the resulting problem is still intractable, its optimal cost can be approximated by the methods discussed in Sect. 4. It'd be interesting to see if useful heuristics for planning could be obtained by polynomial approximations that simplify the preconditions rather than the action delete lists. The scheme for deriving admissible heuristics from [HG00] can actually be seen from this perspective.

The automatic derivations of useful admissible heuristics has also been tackled by Prieditis [Pri93]. Prieditis' scheme is based on a set of transformations that generate a large space of relaxations given problems expressed in a version of Strips. This space is then searched for relaxations that produce heuristics that speed up the search in the original problem. He shows that a number of interesting heuristics can be identified in this way. Our work departs from this in that we stick to one particular type of relaxation for all problems. However, an scheme like Prieditis' could be used as an off-line learning component of heuristic search planners that could tune the type of heuristic for the given domain.

A more recent scheme for deriving heuristics is based on the notion of *pattern databases* developed by Culberson and Schaeffer [CS98] and used by Korf for finding optimal solutions to Rubik's Cube [Kor98]. In a problem like the 15-Puzzle, a pattern database can be understood as a table that contains the optimal costs associated with a relaxed (abstracted) state model in which the location of a certain set of tiles are ignored. Since the relaxed state model can have a much smaller size than the original state model, it can be solved optimally by blind search (e.g., breadth-first search). Then the heuristic $h(s)$ of a state $s$ can be obtained by taking the distance from the projection of $s$ to the projection of the goal $G$ in the relaxed state model. If there are several pattern databases, the maximum of these distances is taken instead. The idea of pattern databases is powerful but is not completely general. Indeed, the size of the relaxed state model that arises in planning problems when the values of certain state-variables are ignored, is not necessarily smaller than the size of the original problem ([HH99] mentions the case of the blocks-world). However, the idea applies very well to permutation problems such as sliding tile puzzles and Rubik's cube, and may have application in many planning domains.

Korf and Taylor [KT96] also sketch a theory of heuristics that may have application in domain-independent planning. In the sliding tile puzzles, their idea is to solve a number of 'relaxed' problems in which we only care about disjoint subsets of tiles and in each case we only count the moves of the tiles selected. Then the addition of such counts provides an admissible heuristic for the original problem. As in the case of pattern databases, each of the relaxed problems involves a smaller state model that can be solved by brute force methods. Also as for pattern databases, the approach seems applicable to permutation problems but not to arbitrary planning problems. In particular, it's not clear how to apply these ideas to a problem like blocks-world.

## 7.3 Heuristic Regression Planning and Graphplan

The operation of the regression planner HSPr consists of two phases. In the first, a forward propagation is used to estimate the costs of all atoms from the initial state $s_0$, and in the second, a regression search is performed using those measures. These two phases are in correspondence with the two operation phases in Graphplan [BF97] where a plan graph is built forward in a first phase, and is searched backward for plans in the second. The two planners are also related in the use of mutexes, and idea that HSPr borrows from Graphplan. For the rest, HSPr and Graphplan look quite different. However, Graphplan can also be understood as an heuristic search planner with a precise heuristic function and search algorithm. From this point of view, the main innovation in Graphplan is the implementation of the search that takes advantage of the plan graph and is quite efficient, and the derivation of the heuristic that makes use of the mutex information. More precisely, from the perspective of heuristic search planning, the main features of Graphplan can be understood as follows:

1. **Plan Graph:** The plan graph encodes an admissible heuristic $h_G$ where $h_G(\mathsf{s}) = j$ iff $j$ is the index of the first level in the graph that includes $\mathsf{s}$ without a mutex and in which $\mathsf{s}$ is not memoized (memoizations are updates on the heuristic function; see 4). The heuristic $h_G$ is a refined version of the $h_{max}$ heuristic discussed in Sect. 4, and is closely related to the family of admissible heuristics formulated in [HG00].

2. **Mutex:** Mutexes are used to prune states in the regression search (as in HSPr) and to refine the heuristic $h_{max}$. In particular, the cost of a set of atoms $C$ is no longer given by the cost of the most costly atom in the set when in the first layer that contains $C$, $C$ occurs with a mutex.

3. **Algorithm:** The search algorithm is a version of Iterative Deepening A* (IDA*) [Kor85], where the *sum* of the accumulated cost $g(n)$ and the estimated cost $h_G(n)$ is used to prune nodes $n$ whose cost exceed the current threshold. Actually Graphplan never generates such nodes. The algorithm takes advantage of the information stored in the plan graph and converts the search in a 'solution extraction' procedure.

4. **Memoization:** Memoizations are updates on the heuristic function $h_G$ (see 1). The resulting algorithm is a memory-extended version of IDA* that closely corresponds to the MREC algorithm [SB89]. In MREC, the heuristic of a node $n$ is updated and stored in a hash-table after the search below the children of $n$ completes without a solution (given the current threshold).

5. **Parallelism:** Graphplan, unlike HSPr, searches a parallel regression space. While the branching factor in this search can be very high, Graphplan makes smart use of the information in the graph to generate only the children that are 'relevant' and whose cost does not exceed the current threshold. The branching rule used in Graphplan is made explicit in [HG00].

In [HG00] Graphplan is compared with a pure IDA* planner based on an admissible heuristic equivalent to Graphplan's $h_G$. In sequential domains the planners have a similar performance, but on parallel domains, Graphplan is more than an order-of-magnitude faster due to the more efficient IDA* search afforded by the plan graph. The plan graph, however, restricts Graphplan to IDA* searches, and it cannot be easily adapted to best-first searches or WIDA* searches [Kor93]. That's why it is not simple to modify a Graphplan-based planner to produce suboptimal solutions fast, while this is trivial in heuristic search algorithms where a multiplying constant $W > 1$ in the heuristic term of the evaluation function often achieves the desired effect [Pea83]. In addition, in serial problems in which subgoals are mostly independent, like Gripper, the heuristic $h_G$ computed by Graphplan yields poor estimates, and Graphplan-based planners are not likely to do well.

# 8 Conclusions

We have presented a formulation of planning as heuristic search and have shown that simple state-space search algorithms guided by a general domain-independent heuristic produce a family of planners that are competitive with some of the best current planners. We have also explored a number of variations, such as reversing the direction of the search for accelerating node evaluation, and extracting information about propositional invariants for avoiding dead-ends. The planner that showed the most solid performance, however, was the simplest planner, HSP2, based on a best-first forward search, in which atom costs are recomputed from scratch in every state.

Heuristic search planners are related to specialized problem solvers but differ from them in the use of a general declarative language for stating problems and a general mechanism for extracting heuristics. Planners must offer good modeling language for expressing problems in a convenient way, and general solvers for operating on those representations and producing efficient solutions.

A concrete challenge for the future is to reduce the gap in performance between heuristic search planners and specialized problem solvers in domains like the 24-puzzle [KT96], Rubik's cube [Kor98], and Sokoban [JS99]. In addition, for planners to be more applicable to real problems, it is necessary that they handle aspects such as non-boolean variables, action durations, and parallel actions. Planners should be able to accommodate a rich class of scheduling problems, yet very few planners currently have such capabilities, and even fewer if any can compete with specialized solvers. Three issues that we believe must be addressed in order to make heuristic search planners more general and more powerful are the ones discussed below.

- **Heuristics:** the heuristics $h_{add}$ and $h_{max}$ considered in this paper are poor estimators, and cannot compete with specialized heuristics. The heuristic $h_G$ used in Graphplan is better than $h_{max}$ but it is not good enough for problems like Rubik's Cube or the 24-puzzle where subgoals interact in complex ways. In [HG00], a class of admissible heuristics $h^m$ are formulated in which the cost of a set of atoms $C$ is approximated by the cost of the mostly costly subset of size $m$. For $m = 1$, $h^m$ reduces to the $h_{max}$ heuristic, and for $m = 2$, $h^m$ reduces to the Graphplan heuristic. Higher order heuristics for $m > 2$, may prove effective in complex problems such as the 24-puzzle and Rubik's cube, and they may actually be competitive with the specialized heuristics used for those problems [KT96, Kor98]. As mentioned in [HG00], the challenge is to compute such heuristics reasonably fast, and to use them with little overhead at run time. Such higher-order heuristics are related to pattern databases but they are applicable to all planning problems and not only to permutation problems.

- **Branching rules:** in highly parallel domains like rockets and logistics, SAT approaches appear to perform best among optimal parallel planners. This may be due to the branching scheme used. In

SAT formulations, the space is explored by setting the value of any variable at any time point, and then considering each of the resulting state partitions separately. In Graphplan and in heuristic search approaches, the splitting is done by applying all possible actions. Yet alternative branching schemes, are common in heuristic branch-and-bound search procedures [LRK85], in particular, in scheduling applications [CP89]. Work on parallel planning, in particular involving actions of different durations, would most likely require such alternative branching schemes.

- **Modeling Languages:** all the planners discussed in this paper are Strips planners. Yet few real problems can actually be encoded efficiently in Strips. This has motivated the development of extensions such as ADL [Ped89] and Functional Strips [Gef99]. From the point of view of heuristic search planning, the issue becomes the derivation of good heuristics from such richer languages. The ideas considered in this paper do not carry directly to such languages but it seems that it should be possible to exploit the richer representations for extracting better heuristics.

# 9 Acknowledgments

We thank Daniel Le Berre, Rina Dechter, Patrik Haslum, Joerg Hoffman, Rao Kambhampati, and Richard Korf for discussions related to this work. This work has been partially supported by grant S1-96001365 from Conicit, Venezuela and by the Wallenberg Foundation, Sweden. Blai Bonet is currently at UCLA with a USB-Conicit fellowship.

# References

[AMO93] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice-Hall, 1993.

[ASW98] C. Anderson, D. Smith, and D. Weld. Conditional effects in graphplan. In *Proceedings AIPS-98*. AAAI Press, 1998.

[BC99] P. Van Beek and X. Chen. CPlan: a constraint programming approach to planning. In *Proceedings AAAI-99*, 1999.

[BF97] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, pages 281–300, 1997.

[BG99] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of ECP-99*. Springer, 1999.

[BK98] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning, 1998. Submitted.

[BLG97] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pages 714–719. MIT Press, 1997.

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job shop problem. *Management Science*, 35(2), 1989.

[CS98] J. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):319–333, 1998.

[FN71] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120, 1971.

[Gef99] H. Geffner. Functional strips: a more general language for planning and problem solving. Logic-based AI Workshop, Washington D.C., 1999.

[HG95] W. Harvey and M. Ginsberg. Limited discrepancy search. In *Proceedings IJCAI-95*, pages 607–613. Morgan Kaufmann, 1995.

[HG00] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Proceedings AIPS-2000*. AAAI Press, 2000. To Appear.

[HH99] R. Holte and I. Hernadvolgyi. A space-time tradeoff for memory-based heuristics. In *Proceedings AAAI-99*, pages 704–709. Mit Press, 1999.

[HSD92] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1–3):113–159, 1992.

[JS99] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *Proceedings IJCAI-99*. Morgan Kaufmann, 1999.

[KNHD97] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proc. 4th European Conf. on Planning*, volume LNAI 1248. Springer, 1997.

[Kor85] R. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kor90] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[Kor93] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.

[Kor98] R. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of AAAI-98*, pages 1202–1207, 1998.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, pages 1194–1201, 1996.

[KS99] H. Kautz and B. Selman. Unifying SAT-based and Graph-based planning. In *Proceedings IJCAI-99*. Morgan Kaufmann, 1999.

[KT96] R. Korf and L. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of AAAI-96*, pages 1202–1207. MIT Press, 1996.

[LF99] D. Long and M. Fox. The efficient implementation of the plan-graph. *JAIR*, 10:85–115, 1999.

[LRK85] E. Lawler and A. Rinnooy-Kan, editors. *The Traveling Salesman Problem : A Guided Tour of Combinatorial Optimization*. Wiley, 1985.

[McD96] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proceedings AIPS-96*, 1996.

[McD98a] D. McDermott. AIPS-98 Planning Competition Results. `http://ftp.cs.yale.edu/pub-/mcdermott/aipscomp-results.html`, 1998.

[McD98b] D. McDermott. PDDL – the planning domain definition language. Available at `http://ftp.cs.yale.edu/pub/mcdermott`, 1998.

[McD99] D. McDermott. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159, 1999.

[Nil80] N. Nilsson. *Principles of Artificial Intelligence.* Tioga, 1980.

[NS63] A. Newell and H. Simon. GPS: a program that simulates human thought. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. McGraw Hill, 1963.

[NS72] A. Newell and H. Simon. *Human Problem Solving.* Prentice–Hall, Englewood Cliffs, NJ, 1972.

[Pea83] J. Pearl. *Heuristics.* Morgan Kaufmann, 1983.

[Ped89] E. Pednault. ADL: Exploring the middle ground between Strips and the situation calculus. In *Proceedings KR-89*, pages 324–332, 1989.

[Pri93] A. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.

[RV99] I. Refanidis and I. Vlahavas. GRT: A domain independent heuristic for Strips worlds based on greedy regression tables. In *Proceedings of ECP-99.* Springer, 1999.

[SB89] A. Sen and A. Bagchi. Fast recursive formulations for BFS that allow controlled used of memory. In *Proceedings IJCAI-89*, pages 297–302, 1989.

[Wel94] D. Weld. An introduction to least commitment planning. *AI Magazine*, 1994.