Planning as Heuristic Search: New Results

Blai Bonet and Héctor Geffner

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

Abstract. In the recent AIPS98 Planning Competition, the HSP planner, based on a forward state search and a domain-independent heuristic, showed that heuristic search planners can be competitive with state of the art Graphplan and Satisfiability planners. HSP solved more problems than the other planners but it often took more time or produced longer plans. The main bottleneck in HSP is the computation of the heuristic for every new state. This computation may take up to 85% of the processing time. In this paper, we present a solution to this problem that uses a simple change in the direction of the search. The new planner, that we call HSPr, is based on the same ideas and heuristic as HSP, but searches backward from the goal rather than forward from the initial state. This allows HSPr to compute the heuristic estimates only *once*. As a result, HSPr can produce better plans, often in less time. For example, HSPr solves each of the 30 logistics problems from Kautz and Selman in less than 3 seconds. This is two orders of magnitude faster than BLACKBOX. At the same time, in almost all cases, the plans are substantially smaller. HSPr is also more robust than HSP as it visits a larger number of states, makes deterministic decisions, and relies on a single adjustable parameter than can be fixed for most domains. HSPr, however, is not better than HSP accross all domains and in particular, in the blocks world, HSPr fails on some large instances that HSP can solve. We discuss also the relation between HSPr and Graphplan, and argue that Graphplan can also be understood as a heuristic search planner with a precise heuristic function and search algorithm.

1 Introduction

The last few years have seen a number of promising new approaches in Planning. Most prominent among these are Graphplan [BF95] and Satplan [KS96]. Both work in stages by building suitable structures and then searching them. 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 [McD98], 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 [BG98], was based on heuristic search [Nil80,Pea83]. In HSP, the search is assumed to be similar to the search in problems like the 8-puzzle, the sole 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 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. Similar ideas have been used recently in [McD96] and [BLG97].

In this paper, we review the heuristic and search algorithm used in HSP, discuss the problem of having to compute the heuristic from scratch in every new state, and introduce a reformulation of HSP that avoids this problem. We also analyze the relation between the resulting planner and Graphplan, and argue that Graphplan can also be understood as an heuristic search planner with a precise heuristic and search algorithm.

2 HSP: Heuristic Search Planner

HSP casts planning problems as problems of heuristic search. A Strips problem $P = \langle A, O, I, G \rangle$ is a tuple 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, and each has precondition, add, and delete lists denoted as $Prec(op)$, $Add(op)$, and $Del(op)$, given by sets of atoms from A. A Strips problem $P = \langle A, O, I, G \rangle$ defines a state-space $\mathcal{S} = \langle S, s_0, S_G, A(\cdot), f, c \rangle$ 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 $Prec(op) \subseteq s$
S5. the transition function f maps states s into states $s' = f(s, a)$, such that $s' = s - Del(a) + Add(a)$ for $a \in A(s)$.

HSP searches this state-space, starting from s_0, with the aid of an heuristic extracted from the Strips representation of the problem (see also [McD96,BLG97]).

2.1 Heuristic

The heuristic function h for a problem P 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 P' is a lower bound on the optimal cost $h^*(s)$ for solving P. The heuristics $h(s)$ could thus be set to $h'(s)$. Solving optimally the relaxed problem P' and hence obtaining $h'(s)$, however, is still exponentially hard. We thus settle for an approximation: we set $h(s)$ to an *estimate* of the optimal value function $h'(s)$ of the relaxed problem. This estimate is computed as follows.

For a given state s and every atom p we compute a measure $g(p)$ that provides an estimate of the cost (number of steps) of achieving p from s. These measures are initialized to 0 if $p \in s$ and to ∞ otherwise. Then, every time an operator op with preconditions $C = Prec(op)$ is applicable in s, each atom $p \in Add(op)$ is added to s and $g(p)$ is updated to

$$g(p) := \min \left[\ g(p)\ ,\ 1 + g(C)\ \right] \tag{1}$$

where $g(C)$ stands for the cost of achieving the *conjunction* of atoms C (see below). The expansions and updates continue until the measures $g(p)$ do not change. This is basically a shortest path algorithm over the graph that connects action preconditions with their effects.

In HSP, the costs $g(C)$ for conjunctions of atoms C is defined as the *sum* of the costs $g(r)$ of the individual atoms r in C:

$$g(C) \stackrel{\text{def}}{=} \sum_{r \in C} g(r) \quad \text{(additive costs)} \tag{2}$$

The heuristic function $h(s)$ that provides an estimate of the number of steps required to make the goal G true from s, is then defined as:

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

Note that the g measures depend on s, which is the state from which they are computed. In HSP, these measures are recomputed from scratch in every new state. This is expensive and may take up to 85% of the processing time.

The definition of the cost of *conjunctions* in (2) assumes that 'subgoals' are *independent*. Namely, that plans that achieve one subgoal have no effect on the plans that achieve other subgoals. In general, this is not true, and as a result the heuristic function defined by 1–3 is not *admissible* (it may overestimate the true optimal costs).

An admissible heuristic function *could* be defined by simply changing the formula (2) to

$$g(C) = \max_{r \in C} g(r) \quad \text{(max costs)} \tag{4}$$

The heuristic function that would result from this change, that we call the 'max heuristic', is admissible but is less informative than the *additive* function defined above. This is the reason that it is not used in HSP. 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. 5 we will see that a refined version of the 'max heuristic' is used in Graphplan.

2.2 Algorithm

Provided with the heuristic h, HSP approaches planning as a problem of heuristic search. While standard search algorithms like A* or IDA* [Kor93] could be used, HSP uses a variation of hill-climbing. This is due to several reasons, the most important being that the computation of $h(s)$ for *every* new state is expensive.

HSP thus tries to get to the goal with as few node evaluations as possible. Surprisingly, the hill-climbing search plus a few simple enhancements, like a limited number of 'plateau' moves, restarts, and memory of past states, performs relatively well. We have obtained similar results before using Korf's LRTA* [Kor90] in place of hill-climbing [BLG97]. This is probably due to the fact that the heuristic h, while not admissible, is informative and tends to drive the search in a good direction.

Table 1 from [McD98], shows the results for the Strips track of the recent AIPS Competition. As can be seen, HSP solved more problems than the other planners but it often took more time or produced longer plans (see [McD98] for details).

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 Competition (Strips track). Columns show the number of problems solved by each planner, and the number of problems in which each planner was fastest or produced shortest plans (from McDermott)

3 HSPr: Heuristic Regression Planning

In HSP, the main bottleneck is the computation of the heuristic in every new state. The main innovation in HSPr is a simple change in the direction of the search that avoids this problem. As a result, the search proceeds faster, more states can be visited, and often better plans can be found in less time.

HSPr searches backward from the goal rather than forward from the initial state. This is an old idea in planning known as *regression search* [Nil80,Wel94]. In regression search, the states can be thought as *subgoals;* i.e., the 'application' of an action in a goal yields a situation in which the execution of the action achieves the goal. The crucial point in HSPr is that the measures $g(p)$ computed from s_0 that estimate the cost of achieving each atom p from s_0 (Sect. 2.1), can be used with *no recomputation* to estimate the heuristic of *any* state arising during the regression. For example, if $s = \{p, q, r\}$ is one such state, the cost of reaching s_0 from s by regression can be estimated in terms of the costs $g(p)$, $g(q)$, and $g(r)$ of achieving p, q, r from s_0. The same applies to any other state. In other words, in HSPr, the cost estimates $g(p)$ are computed *once* from s_0 and are used to define the heuristic of *any* state. This is the key change from HSP from HSPr which is formalized below.

3.1 State space

HSPr and HSP search two different state spaces. HSP searches the *progression space* $\mathcal{S}$ defined by [S1]–[S5] above, where the actions are the available Strips operators. HSPr, on the other hand, searches a *regression space* $\mathcal{R}$ where the actions correspond to 'reverse' application of the Strips operators. The regression space $\mathcal{R}$ associated with a Strips problem $P = \langle A, O, I, G \rangle$ can be defined in analogy to the progression space $\mathcal{S}$ defined by [S1]–[S5] above, as the tuple $\langle \mathsf{S}, \mathsf{s}_0, \mathsf{S}_G, \mathsf{A}(\cdot), \mathsf{f}, \mathsf{c} \rangle$ where

R1. the states s are sets of atoms from A
R2. the initial state s_0 is the goal G
R3. the goal states $\mathsf{s} \in \mathsf{S}_G$ are such $\mathsf{s} \subseteq I$
R4. the set of actions $\mathsf{A}(\mathsf{s})$ applicable in s are the operators $op \in O$ that are *relevant* and *consistent;* namely, $Add(op) \cap \mathsf{s} \neq \emptyset$ 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 $\mathsf{s}' = \mathsf{s} - Add(a) + Prec(a)$.

The solution of this state space is, like the solution of any space $\langle \mathsf{S}, \mathsf{s}_0, \mathsf{S}_G, \mathsf{A}(\cdot), \mathsf{f}, \mathsf{c} \rangle$, a finite sequence of actions a_0, a_1, ..., a_n such that for a sequence of states s_0, s_1, ..., 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.

3.2 Heuristic

HSPr searches the regression space [R1]–[R5] using an heuristic based on the additive cost estimates $g(p)$ described in Sect 2.1. *These estimates are computed only once from the initial state* $s_0 \in \mathcal{S}$. The heuristic $h(\mathsf{s})$ associated with *any* state s is then defined as

$$h(\mathsf{s}) = \sum_{p \in \mathsf{s}} g(p)$$

While in HSP, the heuristic $h(s)$ combines the cost estimates $g(p)$ of a fixed set of goal atoms computed from each state s, in HSPr, $h(\mathsf{s})$ combines the estimates $g(p)$ computed from a fixed state s_0 over the set of subgoals $p \in \mathsf{s}$.

3.3 Mutexes

The regression search often leads to states s that are not reachable from the initial state s_0 in $\mathcal{S}$ and cannot lead to a solution. Graphplan [BF95] recognized this problem and showed a way to deal with it based on the notion of *mutual exclusivity relations* or *mutexes.* In HSPr, we use the same idea with a slightly different formulation.

Roughly, two atoms p and q form a mutex $\langle p,q\rangle$, when they cannot be both true in a state reachable from s_0. For example, in block problems, atoms like $on(a,b)$ and $on(a,c)$ will normally form a mutex. States containing mutex pairs can be safely pruned as they cannot be part of a solution. We are thus interested in recognizing as many mutexes as possible.

A tentative definition is to have a pair of atoms $\langle p,q\rangle$ as a mutex when p and q are not both true in s_0 and every action that asserts p deletes q, and every action that asserts q deletes p. This definition, however, while sound, is too weak. In particular, it does not recognize a pair like $\langle on(a,b), on(a,c)\rangle$ 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 $\langle p,q\rangle$ is recognized as mutex when the actions that add p and do not delete q can guarantee through their preconditions that q will not be true after the action (symmetrically for actions that add q). For that we consider *sets* of mutexes.

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 $\langle p,q\rangle$ in M*

1. *p and q are not both true in s_0,*
2. *for every $op \in O$ that adds p, $q \in Del(op)$, or $q \notin Add(op)$ and for some $r \in Prec(op)$, $\langle r,q\rangle \in M$, and*
3. *for every $op \in O$ that adds q, $p \in Del(op)$, or $p \notin Add(op)$ and for some $r \in Prec(op)$, $\langle r,p\rangle \in M$.*

It is easy to verify that if $\langle p,q\rangle$ belongs to a mutex set, then p and q are really mutually exclusive, i.e., no reachable state will contain them both. 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 $\langle p,q\rangle$ is a 'bad pair' in M when it does not comply with one of the conditions 1–3 above. The procedure for approximating the largest 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 can be chosen as the set of all pairs $\langle p,q\rangle$ for atoms p and q. In practice, however, we've found that we could speed up the computation without losing mutexes by considering a smaller initial set defined as the union of two sets M_A and M_B where

- M_A is the set of pairs $\langle p,q\rangle$ such that for some $op \in O$, $p \in Add(op)$ and $q \in Del(op)$,
- M_B is the set of pairs $\langle r,q\rangle$ such that for some $\langle p,q\rangle$ in M_A and some $op \in O$, $r \in Prec(op)$ and $p \in Add(op)$

The structure of this definition mirrors that of Def. 1. We treat pairs symmetrically and hence assume that $\langle q,p\rangle$ is in a set when $\langle p,q\rangle$ is, and vice versa (in the implementation, atoms are ordered so only one pair is stored).

A mutex in HSPr refers to a pair in the set M^* obtained from $M_0 = M_A + M_B$ 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 the domains we have considered appears to prune the obvious unreachable states. A difference with Graphplan is that this definition identifies non-temporal mutexes while Graphplan identifies time-dependent mutexes. On the other hand, because of the fixed point construction, it 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.[1]

3.4 Algorithm

The search algorithm in HSPr uses the heuristic h to guide the search in the regression space and the mutex set M^* to prune states. The choice of the actual search algorithm follows from several considerations:

1. We want HSPr to solve problems with large state-spaces (e.g., spaces of 10^{20} states are not uncommon).
2. Node generation in HSPr, while faster than in HSP, is slow in comparison with domain-specific search algorithms.[2]
3. The heuristic function, while often quite informative, is not admissible and often overestimates the true optimal costs.
4. State spaces in planning are quite redundant with many paths leading to the same states.

These constraints are common to HSP where node generation is several times slower because of the computation of the heuristic. Slow node generation with large state-spaces practically rules out optimal search algorithms such as A* and IDA* that visit too many states before finding a solution. HSP uses instead a Hill-Climbing search with a few extensions like a memory of past states, a limited number of 'plateau' moves, and multiple restarts. In HSPr, we wanted to take advantage of the faster node generation to use a more systematic algorithm. After some experimentation, we settled on a simple algorithm that we call *Greedy Best First*. GBFS is a BFS algorithm that uses a cost function $f(n) = g(n) + W \cdot h(n)$, where $g(n)$ is the accumulated cost (number of steps),[3] $h(n)$ is the estimated cost, and $W \geq 1$ is a real parameter.[4] As any BFS algorithm, GBFS maintains the nodes for expansion in an OPEN list, and the nodes already expanded in a CLOSED

[1] An additional distinction is that Graphplan focuses on parallel macro-actions, while we focus on sequential primitive actions. See Sect. 5.

[2] HSPr generates a few thousand nodes per second while Korf's algorithms for the Rubik's Cube and the 24-puzzle generate in the order of a million nodes per second [Kor98,KT96].

[3] Not to be confused with the cost measures $g(p)$ defined in Sect. 2.1 for *atoms.*

[4] For the role of W in BFS, see [Kor93].

list. The only difference between GBFS and BFS is that before selecting a node n with minimum f-cost from OPEN, GBFS checks first to see whether the f-best children of the *last node* expanded are 'good enough'. If so, GBFS selects one such child; else it selects n from OPEN as BFS. In HSPr, a child n' of a node n is 'good enough' when it appears closer to the goal; i.e., when $h(n') < h(n)$. GBFS thus performs some sort of Hill-Climbing search but 'backtracks' in a BFS mode when the heuristic is not improved. We also explored schemes in which 'discrepancies' are counted and used to control a LDS algorithm [HG95] but didn't get as good results. The results reported below for HSPr are all based on the GBFS algorithm described above with $W = 5$. Small changes in W have little effect in HSPr, but some problems are not solved with $W = 1$.

4 Experiments

We report results on two domains for which there is a widely used body of difficult but solvable instances. We use the 30 logistic instances and the 5 largest block-world instances included in the BLACKBOX distribution (that we refer to as log-i and bw-k, $i = 1, \ldots, 30$, $k = 1, \ldots, 5$),[5] and 5 hard block-world instances of our own (that we refer to as bw-i, $i = 6, \ldots, 10$).

In logistics, we compare HSPr with HSP, BLACKBOX, and TLPLAN. TLPLAN [BK98] is not a domain-independent planner (it uses control information tailored for each domain) but provides a reference for the results achievable in the domain. The results are shown in Table 2. The 'time' column measures CPU time in seconds, while 'steps' displays the number of actions in the solutions found.[6]

As the table shows, HSPr solves each of the 30 logistics problems in less than 3 seconds. This is faster (and less erratic) than HSP, and two orders of magnitude faster than BLACKBOX. At the same time, in almost all cases, the plans produced by HSPr are substantially smaller than those produced by BLACKBOX and HSP. The speed of HSPr is comparable with TLPLAN that includes fine-tuned knowledge for the domain. The plans obtained by TLPLAN, however, are shorter (with one exception).

HSPr is not better than HSP accross all domains, and there are problems, including a number of 'grid', 'mystery' and 'mprime' instances from the AIPS

[5] The logistics instances are from the 'logistics-strips' directory; while the block-world instances are from the prodigy-bw directory (they contain 7, 9, 11, 15, and 19 blocks, respectively).

[6] The results for BLACKBOX and TLPLAN were taken from [BK98]. The results for BLACKBOX in [BK98] are compatible with the results in the distribution but are slightly more detailed. From [BK98], BLACKBOX was run on a SPARC Ultra 2 with a 296MHz clock and 256M of RAM, while TLPLAN was run on a Pentium Pro with a 200MHz clock and 128M of RAM. The results for HSP and HSPr were obtained on a SPARC Ultra 5 running at 333MHz with 128M of RAM. In the case of HSP and HSPr, individual problems are converted into C programs that are then compiled and run. This takes on the order of a couple of seconds for each instance. This time is not included in the table.

Problem	Time				Steps			
	HSPr	HSP	BBOX	TLPLAN	HSPr	HSP	BBOX	TLPLAN
log-01	0.09	0.63	0.57	0.26	27	39	25	25
log-02	0.08	0.48	95.97	0.28	28	31	31	27
log-03	0.08	0.47	98.99	0.24	29	27	28	27
log-04	0.23	0.95	130.74	1.37	67	58	71	51
log-05	0.19	1.12	231.93	1.10	51	53	69	42
log-06	0.33	1.51	321.27	1.91	69	60	82	51
log-07	1.12	—	264.04	5.54	81	—	96	70
log-08	1.49	9.43	317.42	6.84	82	170	110	70
log-09	0.76	—	1609.45	3.79	77	—	121	70
log-10	0.99	4.77	84.04	2.42	46	109	71	41
log-11	0.64	1.61	137.93	2.24	54	47	68	46
log-12	0.43	1.30	136.22	1.93	41	36	49	38
log-13	1.26	5.78	165.84	6.54	74	102	85	66
log-14	1.46	8.48	77.74	9.34	82	141	89	73
log-15	1.10	3.53	424.36	5.36	69	76	91	63
log-16	0.34	1.09	926.96	1.14	44	43	85	39
log-17	0.36	—	758.47	1.24	48	—	83	43
log-18	2.36	5.88	152.35	9.27	56	57	105	46
log-19	0.74	2.97	149.22	2.66	50	68	78	45
log-20	1.77	7.46	538.22	10.18	99	144	113	89
log-21	1.51	5.09	190.49	6.83	69	81	87	59
log-22	1.22	4.12	846.84	6.40	87	99	111	75
log-23	0.95	—	173.93	4.69	70	—	85	62
log-24	1.05	5.00	74.83	4.71	73	113	87	64
log-25	1.04	—	73.99	4.09	67	—	84	57
log-26	0.80	3.66	233.40	3.64	52	77	80	55
log-27	1.08	4.45	145.16	5.52	76	115	97	70
log-28	2.96	—	867.34	14.53	88	—	118	74
log-29	2.40	8.40	89.51	5.99	50	105	84	45
log-30	0.89	—	495.37	3.42	52	—	92	51

Table 2. Performance over `logistics` problems. A dash indicates that the planner was not able to solve the problem in 10 mins or 100 Mb.

		HSPr		HSP	
Problem	Blocks	Time	Steps	Time	Steps
bw-01	7	0.07	6	0.16	6
bw-02	9	0.15	6	0.59	10
bw-03	11	0.33	10	0.77	12
bw-04	15	8.11	18	5.88	19
bw-05	19	6.96	18	15.80	28
bw-06	8	0.12	19	0.42	12
bw-07	10	1.10	13	0.65	12
bw-08	12	3.73	14	1.97	18
bw-09	14	–	–	3.83	17
bw-10	16	–	–	12.63	20

Table 3. Performance over `blocks-world` problems. A dash indicates that a planner was not able to solve the problem in 10 mins or 100 Mb.

competition (see [McD98]), were neither planner does well. In the blocks-world, in particular, HSP appears to do better than HSPr over some hard instances (bottom of Table 3). This appeared surprising to us as HSP and HSPr use similar heuristics and HSPr can visit many more nodes than HSP. The problem we found is that the additive heuristic may not work as well for HSPr as for HSP due to the changes in the *size* of the states during the regression search. Since $h(\mathsf{s})$ is defined as the sum of the costs $g(p)$ for $p \in \mathsf{s}$, this means that the heuristics favors smaller states over large states. This is not necessarily bad but often makes the definition of 'good children' in the Greedy BFS algorithm inadequate. Indeed, sometimes perfectly good 'children' do not appear closer to the goal than their 'parents' simply because they contain more atoms. We are currently exploring possible ways around this problem. This problem does not happen in HSP because in the forward search all states s have the same size. Namely, atoms that are not in s can be safely assumed to be false then. The same is not true for the regression search.

5 Related Work

HSPr and HSP are descendants of the real-time planner ASP reported in [BLG97]. All three planners perform planning as heuristic search and use the same heuristic but search in different directions or with different algorithms. UNPOP is a planner based on similar ideas developed independently by Drew McDermott [McD96].

The search in HSPr can be understood as consisting of two phases. In the first, a forward propagation is used to compute the measures $g(p)$ that estimate the cost of achieving each of the atoms p from the initial state s_0; in the second, a regression search is performed using those measures. These two phases are in correspondence with the two phases in Graphplan [BF95], where a plan graph is built forward in the first phase, and is searched backward in the second. The two planners are also related in the use of mutexes that HSPr borrows from Graphplan. For the rest, HSPr and Graphplan look quite different. However, we argue below that Graphplan, like HSPr, can also be understood as a 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 algorithm, that is quite efficient, and the derivation of the heuristic function making use of the mutex information. Basically, we argue that the main features of Graphplan can be understood as follows:

1. **Graph:** The graph encodes an admissible heuristic h_G that is a refined version of the 'max' heuristic discussed in Sect. 2.1, where $h_G(\mathsf{s}) = j$ iff j is the index of the first level in the graph that contains (the set of atoms) s without a mutex, and in which s is not memoized (note that memoizations are updates on the heuristic function; see 4).
2. **Mutex:** Mutexes are used to prune states in the regression search (as in HSPr) and to refine the 'max' heuristic into h_G
3. **Algorithm:** The search algorithm is a version of Iterative Deepening A* (IDA*) [Kor93], 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.
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 searches a regression space $\mathcal{R}_p$ that is slightly different than $\mathcal{R}$ above, in which the actions are macro-actions (parallel-actions) composed of compatible primitive actions. Solutions costs in $\mathcal{R}_p$ are given by the number of macro-actions used (the number of time steps). The heuristic $h_G(\cdot)$ is admissible in this space as well, where it provides a *tighter* lower bound than in $\mathcal{R}$. While the branching factor in $\mathcal{R}_p$ can be very high (the number of macro-actions applicable in a state), 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.

This interpretation can be tested by extracting the heuristic h_G from the graph and plugging it into a suitable version of IDA* (MREC) running over the parallel-regression space $\mathcal{R}_p$. The same care as Graphplan has to be taken for generating the applicable 'parallel' actions in each state, avoiding redundant or unnecessary effort. Our claim is that the performance of the resulting algorithm should be close to the basic Graphplan system [BF95] except for a small constant factor.

We haven't done this experiment ourselves but hope others may want to do it. If the account is correct, it will show that Graphplan, like HSP and HSPr, is best understood as a heuristic search planner. This would provide a different perspective to evaluate the strengths and limitations of Graphplan, and may suggest ways in which Graphplan can be improved. For another perspective on Graphplan, see [KLP97].

6 Discussion

We have presented a reformulation of HSP that makes 'planning as heuristic search' more competitive with state of the art planners. A number of problem and challenges remain open. Among those that we think are the most important are the following:

- **better heuristics:** the 'additive' heuristic used in HSP and HSPr is too simple and often overestimates widely the true costs. The 'max' heuristic is too simple in the other direction, and focuses only on the most difficult subgoals ignoring all others. Better heuristics are needed.
- **node generation:** the node generation rates in HSP and HSPr are orders of magnitude slower than in specialized search algorithms [KT96,Kor98]. Implementations need to be improved substantially if domain-independent planning approaches are to compete with domain-specific search algorithms.
- **use of memory:** in hard problems, HSPr may run out of memory. A number of approaches for searching with limited memory are discussed in the literature but none is currently incorporated in the planners discussed.
- **optimal search:** so far we have been concerned with finding 'reasonable' plans, yet if better *admissible* heuristics can be obtained and node generation rates are improved, it'll be feasible to use standard optimal search algorithms such as IDA*.
- **modeling languages:** HSP and HSPr are tied to the Strips language, yet the ideas can be generalized to more expressive planning languages. In [Gef99], we describe Functional Strips, a language that extends Strips with first-class function symbols, and allows the codification of a number of problems (e.g., Hanoi, 8-puzzle, etc) in a way that mimics more specialized representations. In the current implementation, the heuristic function is given by the user, but we expect to be able to exploit the richer language for extracting better heuristics.

Except for the modeling point, the rest of the issues are common to the problems encountered in the application of domain-specific heuristic search methods [JS99]. Indeed, the only thing that distinguishes planning as heuristic search from classical heuristic search is the use of a general declarative language for encoding problems and getting the heuristic information. The biggest challenge is to make the declarative approach competitive with specialized methods while being more general.

Acknowledgements: Part of this work was done while H. Geffner was visiting IRIT, Toulouse. He thanks J. Lang, D. Dubois, H. Prade, and H. Farreny, for their hospitality and for making the visit possible. Both authors also thank R. Korf and D. Furcy for comments related to this work. Partial funding is due to Conicit. B. Bonet is currently at UCLA under a USB-Conicit fellowship.

The code for HSPr is available at `www.ldc.usb.ve/~hector`.

References

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

[BF95] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, Montreal, Canada, 1995.

[BG98] B. Bonet and H. Geffner. HSP: Planning as heuristic search. `http://www.ldc.usb.ve/~hector`, 1998.

[BK98] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning, 1998. Submitted. At `http://www.lpaig.uwaterloo.ca/~fbacchus`.

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

[Gef99] H. Geffner. Functional strips: a more general language for planning and problem solving. Presented at the Logic-based AI Workshop, Washington D.C., June 1999. At `http://www.ldc.usb.ve/~hector`.

[HG95] W. Harvey and M. Ginsberg. Limited discrepancy search. In *Proc. IJCAI-95*, 1995.

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

[KLP97] S. Kambhampati, E. Lambrecht, and E. Parker. Understanding and extending Graphplan. In *Proc. 4th European Conf. on Planning*, LNAI 1248. Springer, 1997.

[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.

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

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

[Kor98] R. Korf. Finding optimal solutions to 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*.

[KS99] H. Kautz and B. Selman. Unifying SAT-based and Graph-based planning. *Proceedings IJCAI-99*.

[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*, 1999.

[McD96] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proc. Third Int. Conf. on AI Planning Systems (AIPS-96)*, 1996.

[McD98] D. McDermott. AIPS-98 Planning Competition Results. `http://ftp.cs.yale.edu/mcdermott/aipscomp-results.html`,1998.

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

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

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

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