New Admissible Heuristics for Domain-Independent Planning

Patrik Haslum
Dept. of Computer Science
Linköpings Universitet
Linköping, Sweden
pahas@ida.liu.se

Blai Bonet
Depto. de Computación
Universidad Simón Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

Héctor Geffner
ICREA/Universidad Pompeu Fabra
Paseo Circunvalación, #8
Barcelona, Spain
hector.geffner@upf.edu

Abstract

Admissible heuristics are critical for effective domain-independent planning when optimal solutions must be guaranteed. Two useful heuristics are the h^m heuristics, which generalize the reachability heuristic underlying the planning graph, and pattern database heuristics. These heuristics, however, have serious limitations: reachability heuristics capture only the cost of critical paths in a relaxed problem, ignoring the cost of other relevant paths, while PDB heuristics, additive or not, cannot accommodate too many variables in patterns, and methods for automatically selecting patterns that produce good estimates are not known.

We introduce two refinements of these heuristics: First, the additive h^m heuristic which yields an admissible sum of h^m heuristics using a partitioning of the set of actions. Second, the constrained PDB heuristic which uses constraints from the original problem to strengthen the lower bounds obtained from abstractions.

The new heuristics depend on the way the actions or problem variables are partitioned. We advance methods for automatically deriving additive h^m and PDB heuristics from STRIPS encodings. Evaluation shows improvement over existing heuristics in several domains, although, not surprisingly, no heuristic dominates all the others over all domains.

Introduction

Admissible heuristics are critical for effective planning when optimal solutions must be guaranteed. So far, two heuristics have been found useful in domain-independent planning: the h^m heuristics, which generalize the reachability heuristic underlying the planning graph (Blum & Furst 1997), and pattern database heuristics, which generalize domain-specific heuristics used successfully in domains such as the 15-puzzle and Rubik's Cube (Culberson & Schaeffer 1996; Korf & Felner 2002). Both of these heuristics, however, suffer from some serious limitations.

The h^m ($m = 1, 2, \ldots$) family of heuristics is defined by a relaxation in which the cost of achieving a set of atoms is approximated by the cost of the most costly subset of size m. This relaxation is then applied recursively in a reachability computation (Haslum & Geffner 2000). For example, the value of $h^1(\{p\})$, for an atom p, is C if p can be achieved by means of an action a such that for each precondition q of a, $h^1(\{q\}) \leqslant C - cost(a)$, with equality holding for at least one of the preconditions. This recursive focus on the most costly subgoal means that h^1 measures in effect the cost of the "critical path" to each goal. In temporal planning, where the cost of a plan equals the makespan, this is often a fairly good approximation since actions needed to achieve separate goals can often be applied concurrently. But in sequential planning, where the cost of a plan is the *sum* of the costs of all actions in the plan, ignoring the costs of achieving several easier goals in favor of considering only the most difficult goal often results in poor heuristic estimates.

As an example, consider a Blocksworld problem involving n blocks, $b_1, \ldots, b_n$, and the goal of assembling an ordered tower with b_1 on top and b_n at the bottom, starting with all blocks on the table. The optimal solution is simply to stack the $n-1$ blocks that are not on the table in the goal state in correct order, resulting in an optimal plan cost of $n-1$ (assuming all actions have unit cost). But the h^1 estimate for this goal is 1, since each of the $n-1$ subgoals on(b_i, b_{i-1}) can be achieved in a single step. The value of h^2 is 2, since two steps are needed to achieve any pair of goal atoms. In general, h^m will yield a value of m, but since the cost of computing h^m grows exponentially with m, it does not scale up as n grows in this problem.

In cases like this, *adding* the estimated costs of each subgoal (Bonet & Geffner 1999) or *counting* actions in the relaxed plan (Hoffmann & Nebel 2001) tends to give more accurate estimates. However, these simple solutions forfeit admissibility since they fail to take into account the fact that some actions may contribute to the achievement of more than one subgoal. Here, and to our knowledge for the first time, we introduce an *additive* heuristic that is also *admissible*. The idea is simple: Given the heuristic h^m and a partitioning of the set of actions into disjoint sets $A_1, \ldots, A_n$, we define additive h^m as the sum $\sum_i h^m_{A_i}$, where $h^m_{A_i}$ is defined exactly like h^m except that the cost of any action *not* in A_i is considered to be zero. In the example above, for instance, if each set A_i contains the actions that move block b_i, the resulting additive h^1 yields the exact solution cost. This is in fact a general principle which can be applied to any admissible heuristic when the cost of a solution is the sum of individual action costs.

Pattern Databases (PDBs) are memory-based heuristic functions obtained by abstracting away certain problem variables, so that the remaining problem (the "pattern") is small enough to be solved optimally for every state by blind exhaustive search. The results, stored in a table in memory, constitute a PDB for the original problem. This gives an admissible heuristic function, mapping states to lower bounds by mapping the states into "abstract states" and reading their associated value from the table. Recently, Edelkamp (2001) has shown how the idea of PDBs can be applied to domain-independent planning. For effective use of memory, Edelkamp bases patterns on multi-valued variables implicit in many STRIPS problems and makes use of a certain form of independence between such variables which enables estimates from several PDBs to be added together, while still admissible. The implicit variables correspond to sets of atoms that share with every state reachable from the initial situation exactly one atom (the "value" of the variable in that state). Multi-valued variables are thus a kind of invariant, and can be extracted automatically from the STRIPS problem.

For example, in the Blocksworld domain, there are variables `pos(`b`)`, encoding the position of block b, with values `(on `b b'`)`, for every other block b', and `(on-table `b`)`. In the example problem above, of constructing a tower from the n blocks initially on the table, a PDB with variable `pos(`b_i`)` as the pattern yields an estimate of 1, for every block except the bottom-most in the tower. Moreover, the variables are independent, in the sense that every action changes the value of only one variable (every action only moves one block), and therefore the values of these PDBs can be admissibly added, for a total estimate of $n-1$, which is also the optimal cost. Consider now a different problem, in which the blocks are stacked in a single tower in the initial state, again with b_1 on top and b_n at the bottom, and the goal is to reverse the two blocks at the base of the tower, *i.e.* the goal is `(on `b_n b_{n-1}`)`. This goal requires n steps to solve. But the value obtained from a PDB over the pattern `pos(`b_n`)` is 1. This is because two of the preconditions for moving block b_n, that it and the destination are clear, are abstracted away in the computation of this PDB. Including more `pos(`b_i`)` variables in the pattern will not help: this is the best value that can be obtained from a standard PDB built over these variables. However, in the Blocksworld domain there are also variables `top(`b`)`, encoding what is on top of block b, with values `(on `b' b`)`, for every other block b', and `(clear `b`)`. These variables are not additive with each other, nor with the position variables. A PDB for a pattern consisting of the two variables `top(`b_n`)` and `top(`b_{n-1}`)` gives a value of 3; in general, a PDB for a pattern consisting of the "top-of" variables for the k blocks at the base of the tower will yield an estimate of $k+1$ (for $2 \leqslant k < n$).

Because PDB heuristics solve the abstracted problem exhaustively, time and memory constraints prevent patterns from including too many variables. This raises the problem of selecting patterns that will produce good estimates, within given limits, a problem that has been only partially addressed in previous work on PDBs for planning (Holte *et al.* 2004; Edelkamp 2001), yet is critical for the efficient use of PDB heuristics. Additivity is one important criterion, since adding yields better estimates than maximizing, while keeping patterns small, but alone this is not enough. We present two improvements to the basic PDB heuristic for domain-independent planning: the first is "constrained abstraction", where constraints from the original problem are used to prune possible state transitions during the computation of the PDB; the second is a method for selecting variable patterns, under limitation on the size of each individual PDB, by iteratively building PDBs, examining them for weaknesses and extending the pattern with variables that counter them.

Reachability Heuristics

We restrict our attention to propositional STRIPS planning with additive costs. A planning problem P consists of a finite set of atoms, a finite set of actions, an initial state, s_0, represented by the set of all atoms initially true, and a goal description, G, represented by the set of atoms required to be true in any goal state. Each action a is described by three set of atoms: $pre(a)$, $add(a)$ and $del(a)$, known as the precondition, add and delete lists respectively. Associated with every action is a cost, $cost(a)$, and the cost of a plan is the sum of the costs of the actions in the plan. In most benchmark domains the cost is 1 for all actions, *i.e.* the objective is to minimize the number of actions in the plan.

We consider regression planning, in which the search for a plan is carried out backwards from the goal, and associate with a planning problem P a search space (directed graph) $R(P)$. States are subsets of atoms, representing sets of subgoals to be achieved. There is a transition $(s, a, s') \in R(P)$ iff there is an action a such that s regressed through a yields s' (*i.e.* $s \cap del(a) = \emptyset$ and $s' = (s - add(a)) \cup pre(a)$). The cost of the transition equals $cost(a)$. We define $h^*(s)$ to be the minimum cost of any path in $R(P)$ from s to any state *contained* in s_0 (∞ if no path exists). In other words, $h^*(s)$ is the optimal cost of a plan that when executed in s_0 achieves s. Where necessary, we will subscript h^* with an indication of which search space it refers to.

The h^m Heuristics

The h^m ($m = 1, 2, \ldots$) family of heuristics is defined by a relaxation where the cost of a set of atoms is approximated by the cost of the most costly subset of size m. Formally, they can be defined as the solution to

$$h^m(s) = \begin{cases} 0 & \text{if } s \subseteq s_0 \\ \min_{s':(s,a,s') \in R(P)} h^m(s') + cost(a) & \text{if } |s| \leqslant m \\ \max_{s' \subseteq s, |s'| \leqslant m} h^m(s') & o.w. \end{cases} \tag{1}$$

This equation applies the relaxation to the Bellman equation, which characterizes the optimal cost function $h^*(s)$. For small values of m (typically, $m \leqslant 2$), an explicit solution can be computed efficiently by a simple shortest-path algorithm. The solution is stored in the form of a table of heuristic values for sets of m or fewer atoms, so that only $\max_{s' \subseteq s, |s'| \leqslant m} h^m(s')$ needs to be computed when a state s is evaluated during search.

Additive h^m Heuristics

We devise an additive version of the h^m heuristic by introducing a *cost relaxation*: For any subset A of the actions in a planning problem P, the heuristic $h^m_A(s)$ is defined exactly the same as h^m above, except that the cost of every action that does not belong to A is considered to be zero. In other words, h^m_A counts the cost of actions in A, but does not count the cost of actions not in A (we'll say the *costs* of those actions are *relaxed*).

Theorem 1 *Let $A_1, \ldots, A_n$ be disjoint subsets of the actions in planning problem P. Then $h^m_\Sigma(s) = \sum_i h^m_{A_i}(s) \leqslant h^*(s)$ for all s, i.e. $h^m_\Sigma(s)$ is an admissible heuristic for regression planning.*

Proof: The cost of any solution path $\pi = (a_1, \ldots, a_k)$, $\sum_{a_j \in \pi} cost(a_j)$, can be split over the action sets $A_1, \ldots, A_n$, plus the set of actions not in any set A_i: $\sum_{\{a_j \in \pi | a_j \in A_1\}} cost(a_j) + \ldots + \sum_{\{a_j \in \pi | a_j \in A_n\}} cost(a_j) + \sum_{\{a_j \in \pi | a_j \notin \bigcup_i A_i\}} cost(a_j)$, thus $\sum_i h^*_{A_i}(s) \leqslant h^*(s)$. From this follows that $\sum_i h_{A_i}(s) \leqslant h^*(s)$ for any function h that underestimates h^*, including h^m. □

In other words, cost relaxation can be applied to any admissible heuristic for sequential planning, or indeed any setting where the cost of a solution is the sum of individual action costs. Theorem 1 is a special case of the fact that $\sum_i h_{A_i}(s) \leqslant h^*(s)$ holds for any admissible heuristic function h.

Consider again the first Blocksworld problem described in the introduction, of building a tower out of n blocks that are initially all on the table, and suppose actions are partitioned into n sets, such that set A_i contains all actions that move block b_i. The shortest solution for each subgoal $\texttt{on}(b_i, b_{i-1})$ is a single step plan that moves only block b_i to its goal position, so $h^1_{A_i}(\texttt{on}(b_i, b_{i-1})) = 1$, and this is also the value of $h^1_{A_i}(G)$, for G the goal of having all blocks in their goal positions. In exactly the same way, $h^1_{A_i}(\texttt{on}(b_i, b_{i-1})) = 1$, while $h^1_{A_j}(\texttt{on}(b_i, b_{i-1})) = 0$ for every $j \neq i$, since the action moving block b_i is counted as having cost zero in $h^1_{A_j}$. Thus, the sum $h^1_{A_1}(G) + \ldots + h^1_{A_n}(G)$ will be $n - 1$, the optimal plan length.

In practice, the collection of subsets $A_1, \ldots, A_n$ will be a partitioning of the set of actions, so that the cost of every action is counted in some $h^m_{A_i}$. We will return to the problem of how the actions are partitioned after the discussion of pattern database heuristics in the next section.

Pattern Database Heuristics

Pattern database heuristics are based on *abstractions* (or *projections*) of problems: functions mapping states of the search space into a smaller space, with the property that the projection of a solution path in the original space is also a solution in the abstract space (Culberson & Schaeffer 1996).

Abstractions of planning problems are defined as follows: Any subset, A, of the atoms in problem P defines an abstraction P^A, in which the initial and goal states, and the precondition, add and delete lists of each action are simply intersected with A. The problem P^A can be seen as a relaxation of P where only the status of some propositions is important – all atoms that are not in the "pattern" A are abstracted away. We associate with P^A the search space $R(P^A)$ in the same way as for the original problem P. We'll use $h^A(s)$ for the minimum cost of any path in $R(P^A)$ from s to the abstract initial state (*i.e.* $h^A = h^*_{R(P^A)}$): this is a lower bound on the corresponding cost in P, *i.e.* $h^A(s) \leqslant h^*(s)$ for all s. Given that the abstract problem is sufficiently small, the optimal cost function $h^A(s)$ can be computed for all s by means of a breadth-first search in the reversed search space, starting from the projection of s_0. The results are stored in a table, which we'll refer to as the pattern database (PDB) for pattern A. By projecting a state and looking up the cost of the corresponding abstract state, stored in the table, we obtain an admissible heuristic. Immediate from the preceding definitions, we have that for all s

$$\max(h^A(s), h^B(s)) \leqslant h^{A \cup B}(s) \quad (2)$$

for any patterns A and B. Also, under certain conditions this can be strengthened to

$$\max(h^A(s), h^B(s)) \leqslant h^A(s) + h^B(s) \leqslant h^{A \cup B}(s) \quad (3)$$

in which case we say that A and B are *additive*. A sufficient condition for additivity between two patterns, which is also easy to check, is that no action *adds* atoms in both sets.

The memory required to store the PDB, and the time needed to compute it, grows exponentially with the size of the atom set retained by the abstraction. However, as shown by Edelkamp (2001), memory can be used more effectively by basing abstractions on multi-valued variables implicit in the planning problem. Such variables correspond to a particular type of invariant, the property that exactly one from a set of atoms is true in every reachable state. Methods for automatically extracting such "exactly-one-of" invariants from STRIPS encodings have been proposed (*e.g.* Edelkamp & Helmert 1999; Scholz 2000; Rintanen 2000).

Constrained PDBs

Let us return to the second Blocksworld problem described in the introduction: n blocks, $b_1, \ldots, b_n$, are in an ordered tower with b_n at the base and b_1 at the top, and the goal is to swap the two bottom blocks, *i.e.* the goal is $\texttt{(on}\ b_n\ b_{n-1}\texttt{)}$. As stated earlier, a PDB with only variable $\texttt{pos}(b_n)$ as pattern gives an estimate of only 1. This is because the single-step plan moving b_n directly onto b_{n-1} is possible in the abstract space, where everything except the atoms specifying the position of block b_n, including the atoms $\texttt{clear}(b_n)$ and $\texttt{clear}(b_{n-1})$, has been abstracted away from the preconditions of this move action. For the same reason, the estimate is still 1 even if the position of the block immediately above b_n, *i.e.* variable $\texttt{pos}(b_{n-1})$, is also included in the pattern. This, however, is less reasonable, since the value of this variable in the initial state is $\texttt{(on}\ b_{n-1}\ b_n\texttt{)}$, which means block b_n can not be clear. In fact, the two atoms $\texttt{(on}\ b_{n-1}\ b_n\texttt{)}$ and $\texttt{(clear}\ b_n\texttt{)}$ are *mutex*, a fact that the PDB computation fails to take advantage of.

Let $C = \{C_1, \ldots, C_m\}$ be a collection of subsets of the atoms in a STRIPS problem P, such that $|C_i \cap s| \leqslant 1$ for any state s reachable from s_0. In other words, each C_i is an invariant of the 'at-most-one-atom' kind. Static problem mutexes are examples of (binary) invariants of this kind. Given an abstraction of the problem, P^A, we define the search space of the *constrained* abstraction, $R_C(P^A)$, as the subgraph of $R(P^A)$ containing all and only states satisfying all the invariants in C, *i.e.* states s such that $|C_i \cap s| \leqslant 1$ for all $C_i \in C$, and edges (s, a, s') between such states except those corresponding to an action a such that $s' \cup pre(a)$ violates some invariant in C. In other words, the constrained abstraction excludes states that are "impossible" in the original problem, and transitions that are known to be impossible given only the part of the problem kept by the abstraction. We denote the optimal cost function associated with the constrained abstract space $h_C^A(s)$.

Theorem 2 *$h^A(s) \leqslant h_C^A(s) \leqslant h^*(s)$, for all s, i.e. optimal cost in the constrained abstract space is a lower bound on optimal solution cost in the original problem, and it is at least as strong as the lower bound obtained from unconstrained abstraction.*

Proof: A solution path in $R(P)$ corresponds to an action sequence executable in s_0, so it must satisfy all invariants and hence it exists also in the constrained abstract space, as well as in the unconstrained abstraction. □

Thus, PDBs computed under constraints still yield admissible heuristics, provided the constraints are of the "at-most-one-atom" (or "exactly-one-atom") kind. The condition for additivity of PDBs, *i.e.* that no action adds atoms in both patterns, holds also for constrained PDBs.

If we compute the PDB $h_C^{\{\texttt{pos}(b_n),\texttt{pos}(b_{n-1})\}}$, using the static problem mutexes as the set of constraints C, we obtain an estimated cost of 2 for the problem of swapping blocks b_{n-1} and b_n at the bottom of a tower, since b_n must now be cleared before it can be moved. Moreover, if we include also $\texttt{pos}(b_{n-2})$ in the pattern, the estimated cost is 3, since also $(\texttt{clear}\ b_{n-1})$ is a precondition of the move and this is mutex with the position of block b_{n-2}, which is on b_{n-1} in the initial state.

The use of constraints in the computation of PDBs is an improvement over Edelkamp's formulation of PDBs for planning, and makes it possible to derive *e.g.* the sum-of-pairs heuristic for the 15-Puzzle (Korf & Felner 2002): without mutex constraints, a PDB for a pattern consisting of the variables representing the positions of two of the tiles will not give the correct value, since the STRIPS encoding of the problem enforces the condition that two tiles can not occupy the same square by preconditioning actions with the "blank" being in the square moved to, and the position of the blank is not part of the 2-tile pattern.

Using Additive h^m and PDB Heuristics

The additive h^m and PDB heuristics both have "free parameters": for additive h^m the choice of disjoint sets of actions, for PDBs the choice of patterns. The quality of the resulting heuristics often depends very much on these choices, and thus for effective use of the heuristics it is crucial to make them well. Yet, good choices are rarely obvious, and must also be made automatically from the STRIPS formulation. This problem has been addressed only partially in previous work on the use of PDB heuristics for domain-independent planning. Here, we propose methods for the automatic selection of an action partitioning for the h^m heuristic and patterns for the PDB heuristic.

Partitioning Actions

While the sum $\sum_i h_{A_i}^m$ is admissible for any collection $A_1, \ldots, A_n$ of disjoint sets of actions, we should of course choose a partitioning of the actions that yields as high values as possible for the states visited during the search. Our basic approach is to create one partition A_i for each goal atom g_i, and assign actions to the partition where they appear to contribute the most to the sum.

To determine if an action a is relevant for goal atom g_i, we compare the heuristic resulting from relaxing the cost of a to the one that does not: if the value of $h_{A-\{a\}}^1(g_i)$ is less than $h_A^1(g_i)$, then action a belongs to the critical path to g_i, and is clearly relevant. (For efficiency, the h^1 heuristic is used for this test even though it is h^2 that will be computed for the final partitions.) By computing the "loss", $h_A^1(g_i) - h_{A-\{a\}}^1(g_i)$, we also get a (rough) measure of the strength of the relevance of action a to each of the different goals, and the action can be assigned to the goal where relaxing its cost causes the greatest loss. Initially, each of the partitions $A_1, \ldots, A_n$ contains all actions. When an action is "assigned" to one partition it is removed from all the other, so that at the end, when all actions have been assigned, the partitions are disjoint.

The test $h_{A-\{a\}}^1(g_i) < h_A^1(g_i)$, however, is not enough: because h^1 considers, recursively, the most costly atom in any set, the value of $h^1(g_i)$ is in most cases not determined by a single critical path, but rather a "critical tree", branching out on the preconditions of actions. Thus, relaxing the cost of a single action often does not change the h^1 estimate of an atom. To counter this problem, we perform a preliminary division of actions into sets of related actions and perform the test on these sets, *i.e.* instead of checking the effect of relaxing the cost of a single action, we check the effect on the heuristic value of simultaneously relaxing the cost for a set of actions. The preliminary division of actions is based on the same kind of multi-valued variables implicit in the problem as is used for PDB heuristics: we simply find a maximal set of additive variables and let the actions that affect each of the variables form a set. By selecting additive variables, we ensure that the corresponding action sets are disjoint.

When the partitioning of the action set is done, we compute $h_{A_i}^2$ for each partition A_i, as well as the standard h^2. The final heuristic is $\max(h^2, \sum_i h_{A_i}^2)$. Because the cost of both precomputing and evaluating $\sum_i h_{A_i}^2$ dominates that of precomputing and evaluating h^2, there is no motivation not to do the extra maximization step.

Pattern Selection

Basing patterns on the multi-valued variables implicit in a STRIPS problem, and on additive variables in particular, is useful as it allows the PDB to be represented more compactly. Thus, a first approach to using PDB heuristics is to make a pattern out of each variable, sum the values from maximal sets of additive PDBs, and take the maximum of the resulting sums. However, recall that for two additive atom sets, A and B, $h^A(s)+h^B(s) \leqslant h^{A\cup B}(s)$. Thus, while the sum of individual variable PDBs is admissible, it may not be the best heuristic obtainable. The example of disassembling a tower of blocks, discussed in the preceding section illustrates the point. Experimental results also confirm that this example is not at all unusual. On the other hand, it is not always the case that grouping additive variables leads to a better heuristic, and since the size of the PDB $h^{A\cup B}$ equals the product of the sizes of PDBs h^A and h^B, we do not want to use $h^{A\cup B}$ in those cases where $h^A(s) + h^B(s)$ is just as good.

We approach this problem by constructing patterns iteratively. First, we compute a PDB for each variable alone; then we analyze the projected solutions for possible conflicts, and merge the patterns that appear to be most strongly interdependent. This is repeated until no conflicts are found, or no more mergers can be made without making the resulting PDB too large. The process is applied to each set of additive variables separately, so that the resulting collection of PDBs is additive. The final heuristic maximizes over the sums obtained from each such set. All PDBs are computed using the static mutexes of the problem as constraints[1].

To detect conflicts, we make use of the PDB itself: recall that a PDB is computed by an exhaustive breadth-first exploration of the search space in the direction opposite to that of the search for a plan. Thus, a PDB for regression planning is computed by exploring forwards from the initial state. During this exploration, we can store with every abstract state a "backpointer" indicating which action was applied in which (abstract) state to reach this state. Using these, we can for any entry in the PDB quickly extract a corresponding projected plan. Given a pattern A, consisting of a set of variables, and the corresponding PDB, we find conflicts by extracting the projected plan for the goal (projected onto A) and simulating this plan, forwards from the initial state, using the original (unprojected) actions. When we find that a precondition of an action in the projected plan is inconsistent with the value of a variable V not in the pattern, either because the precondition is on the value of V and contradicts the current value or because the precondition is mutex with the value of V, a conflict between pattern A and the pattern that V is part of is noted.

As mentioned, the iterative process is applied to each set of additive variables separately. For each such set of additive variables, $\{V_1, \ldots, V_n\}$, we start with a corresponding collection of patterns $A_1, \ldots, A_n$, and compute for each pattern a PDB and its conflicts with the other patterns. Each conflict is assigned a weight, given by the h^1 value of the inconsistent atom set. A pair of patterns A_i and A_j is a feasible merge iff the size of the PDB for $A_i \cup A_j$ is below a given limit (a parameter of the procedure). We find the feasible pair with the heaviest conflict, merge the two patterns and compute the new PDB and its conflicts, and repeat until there are no more conflicts or no feasible pair to merge. Values from the resulting PDBs are added together, and the maximum over the sums corresponding to each additive variable set is the final heuristic value.

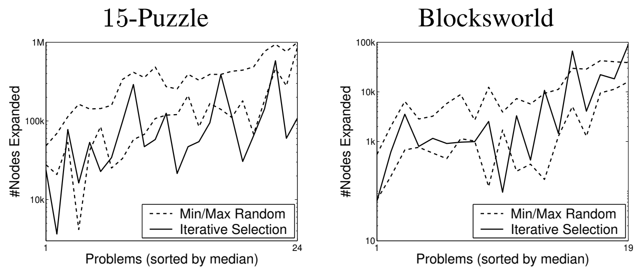

Figure 1: Comparison of PDB heuristic resulting from iterative random pattern selection. The min/max random curves are taken over five different random trials.

[1]Since computing *all* static mutex relations for a planning problem is in general as hard as solving the problem itself, we compute only a sound approximation, *i.e.* a subset, of this set, using essentially the same method as in (Bonet & Geffner 1999).

Experimental Results

As a measure of the *quality* of a heuristic we take the number of node expansions an A* search using the heuristic requires to solve a given problem. The *usefulness* of a heuristic, on the other hand, is determined by the ability of the search to solve problems within given time and/or memory limits using the heuristic, and thus depends also on the time overhead in precomputing and evaluating the heuristic. The additive h^2 and PDB heuristics spend more time precomputing heuristics than standard h^2, and additive h^2 also has a higher computational cost per evaluation.

First, we assess the effectiveness of the iterative pattern selection method, by comparing it to a PDB heuristic which groups variables at random, up to the given limit on the size of each PDB (2 million entries in this experiment). Like the iterative selection, the random grouping is applied to each additive set of variables separately, and the final heuristic is the maximum over the sums thus obtained. We repeat this construction for several (in this experiment, five) different random groupings and consider the best and worst of the heuristics, for each problem. Figure 1 shows the result. There are two conclusions we can draw from this experiment: first, the gap between the best and worst randomly constructed PDB heuristic indicates the potential for improvement of heuristic quality by taking more than just additivity into account when selecting patterns; second, the iterative selection compares fairly well with the result of taking the max of five random PDB heuristics – in the 15-puzzle, it even yields better results for more than half the instances.

Second, we compare the h^2, additive h^2 (as mentioned, the heuristic here referred to as "additive h^2" is in fact

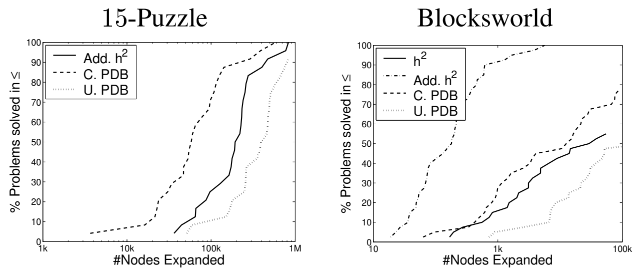

Figure 2: Distribution of the number of node expansions needed to solve a problem using h^2, additive h^2 and Constrained and Unconstrained PDB heuristcis (created by iterative pattern selection).

$\max(h^2, \sum_i h^2_{A_i}))$ and the PDB heuristics (created by iterative pattern selection) with and without mutex constrained projection, over a number of domains, including Blocksworld, a STRIPS encoding of the 15-puzzle, and five domains from the 4th IPC.

Figure 2 shows distributions of the number of node expansions required to solve problems in the Blocksworld and 15-puzzle domains using the different heuristics (instances count as unsolved if requiring more than 1 million nodes, 16 hours CPU time, or 1Gb memory). Clearly, additive h^2 and the constrained PDB heuristic both improve significantly over h^2 and the PDB heuristic without constraints, respectively, in terms of heuristic quality in these domains. Also, the constrained PDB heuristic very clearly outperforms additive h^2 in the 15-puzzle domain, while in Blocksworld it is the other way around. Comparing CPU time instead of node expansions gives a similar picture, except that additive h^2 generally uses more time than unconstrained PDBs in the 15-puzzle, and that h^2 is faster than PDBs for very simple Blocksworld problems. Note that A* search using h^2 fails to solve any 15-puzzle instance within limits.

In the remaining domains, additive h^2 improves over h^2 in terms of quality in the Airport, Promela (philosophers), PSR and Satellite domains, while in the Pipesworld they are equal. Additive h^2 takes more CPU time in all IPC domains, but is still able to solve more problems in the Promela and Satellite domains, where search using h^2 quickly exhausts memory. The constrained PDB heuristic works well in the Satellite domain, where it outperforms additive h^2 for harder instances, in both quality and runtime. In the other IPC domains, however, it performs worse than the other heuristics, even worse than plain h^2.

Conclusions

We have introduced two refinements of existing admissible heuristics for domain-independent planning. The additive h^m heuristic is defined by computing several instances of h^m, each counting the cost of only a subset of the actions in the problem, while relaxing the cost of remaining actions. To our knowledge, this is the first admissible additive alternative to pattern database heuristics for domain-independent planning. Moreover, the principle of cost relaxation is general, and applicable to any admissible heuristic in any setting where the cost of a solution is the sum of individual action costs. For PDB heuristics, we introduced constrained abstraction, which strengthens PDB heuristics by carrying generalized mutex constraints from the original problem into the abstracted problem.

Both PDBs and the additive h^m heuristic have free parameters (the action partitioning and the set of patterns, respectively) which need to be set properly for producing good heuristic estimates. We have proposed methods for selecting these parameters, automatically from the STRIPS encoding. This is a critical problem in the domain-independent setting, which had not been adequately addressed before.

Experimental evaluation shows improvements over existing admissible heuristics, in some case quite significant, though no heuristic dominates all others over all the domains considered. This, however, is not necessary for improving performance of a planner, as the new heuristics need not replace existing ones. They can all be part of the toolbox, and if needed they can be effectively and admissibly combined by maximizing over several alternatives.

References

Blum, A., and Furst, M. 1997. Fast planning through graph analysis. *Artificial Intelligence* 90(1-2):281 – 300.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proc. 5th European Conference on Planning (ECP'99)*, 360 – 372.

Culberson, J., and Schaeffer, J. 1996. Searching with pattern databases. In *Canadian Conference on AI*, volume 1081 of *LNCS*, 402 – 416. Springer.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. 5th European Conference on Planning (ECP'99)*, 135 – 147.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. 6th European Conference on Planning (ECP'01)*, 13 – 24.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 140 – 149. AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of AI Research* 14:253 – 302.

Holte, R.; Newton, J.; Felner, A.; Meshulam, R.; and Furcy, D. 2004. Multiple pattern databases. In *14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, 122 – 131.

Korf, R., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1-2):9 – 22.

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *Proc. 17th National Conference on Artificial Intelligence (AAAI'00)*, 806 – 811.

Scholz, U. 2000. Extracting state constraints from PDDL-like planning domains. In *Proc. AIPS 2000 Workshop on*

Analyzing and Exploiting Domain Knowledge for Efficient Planning, 43 – 48.