Enhancing the Context-Enhanced Additive Heuristic with Precedence Constraints

Dunbo Cai
Jilin University
College of CS and Technology
Qianjin Street 2699
130012 Changchun, China
dunbocai@gmail.com

Jörg Hoffmann
SAP Research
CEC Karlsruhe
Vincenz-Prießnitz-Straße 1
76131 Karlsruhe, Germany
joe.hoffmann@sap.com

Malte Helmert
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee 52
79110 Freiburg, Germany
helmert@informatik.uni-freiburg.de

Abstract

Recently, Helmert and Geffner proposed the context-enhanced additive heuristic, where fact costs are evaluated relative to context states that arise from achieving first a pivot condition of each operator. As Helmert and Geffner pointed out, the method can be generalized to consider contexts arising from arbitrary precedence constraints over operator conditions instead. Herein, we provide such a generalization. We extend Helmert and Geffner's equations, and discuss a number of design choices that arise. Drawing on previous work on goal orderings, we design a family of methods for automatically generating precedence constraints. We run large-scale experiments, showing that the technique can help significantly, depending on the choice of precedence constraints. We shed some light on this by profiling the behavior of all possible precedence constraints, using a sampling technique.

Introduction

Helmert and Geffner (2008) devise a powerful new heuristic for satisficing planning (no optimality guarantee), called the "context-enhanced additive heuristic" h^{cea}. That heuristic is based on a formulation of the additive heuristic h^{add} (Bonet and Geffner 2001) where fact costs are evaluated relative to particular context states. These states arise from achieving first a "pivot condition" of each operator, namely the condition referring to the same variable as the operator's effect. Helmert and Geffner (2008) show that h^{cea} generalizes the causal graph heuristic h^{CG} (Helmert 2004), and that it generally outperforms its competitors, i.e., h^{add}, h^{CG}, and h^{FF} (Hoffmann and Nebel 2001).

As Helmert and Geffner (2008) point out, there is no reason to limit context states to pivot conditions. Intuitively, *the context for a condition q should be taken from another condition p if achieving p may affect the cost of achieving q.* Such dependencies can be captured in the form of precedence constraints (arbitrary partial orders) over the conditions. Helmert and Geffner give the following example. In a Grid-like domain, an operator that "unlocks a door D at a location L with key K" could have the form "locked(D), at(L), have(K) $\rightarrow$ unlocked(D)". If K is currently at a different location L', then the cost of applying this operator should include the cost for moving from L' to L. This corresponds to first achieving "have(K)" and then "at(L)", i.e., we should have the precedence constraint "have(K) $<$ at(L)".

We herein devise what we call the *precedence constraints contexts heuristic* h^{pcc}, which determines precedence constraints in a pre-process to planning, and generalizes h^{cea} to make use of them. In this way, h^{pcc} realizes Helmert and Geffner's idea. We answer several research questions:

How to determine the precedence constraints? A natural answer is to draw on previous work on goal orderings. Koehler and Hoffmann (2000) define a "reasonable order" $p <_r q$ if, to achieve p, one must delete q. Hence it is "reasonable" to achieve p first (so as to avoid having to re-achieve q). For our purposes, achieving p increases the cost of achieving q, which clearly is relevant to the heuristic. Note that "have(K) $<_r$ at(L)" in the above example. We also consider known generalizations (Hoffmann, Porteous, and Sebastia 2004; Richter, Helmert, and Westphal 2008), resulting in a family of 7 automatic pre-processing methods deriving precedence constraints.

How exactly to extend h^{cea}? Extending Helmert and Geffner's equations is non-trivial, and involves a number of design decisions, which we discuss. Most importantly, we identify a possible pathological behavior. Due to the interplay of the approximations made, it may happen that the cost estimate for achieving just the pivot, from the original state, is more than the cost estimate for achieving *all* conditions (including the pivot), from their respective context states. We devise two alternative fixes avoiding this behavior.

How does the extended heuristic perform in practice? We run a large set of experiments evaluating the various parameters of our design, and the performance compared to h^{cea}. It turns out that the scale of the effect on performance is generally not dramatic, below or within 1 order of magnitude. However, the method can help significantly, and leads to improved coverage in several domains. The choice of precedence constraints is highly critical. The strength of our 7 variants varies considerably across, and often even within, domains. To shed some light on this, we sample from the space of all possible sets of precedence constraints. We examine the distribution of search space size. Amongst other things, this provides a picture of "where we are" and what further advantages could potentially be achieved by designing alternative methods for deriving precedence constraints.

Planning Formalism and Notations

Our notations follow those of Helmert and Geffner (2008). A planning *task* is a tuple (V, s_0, s_*, O). Here, V is a set of *variables* v with finite domains D_v; s_0 is the *initial situation*, given as a state over V, i.e., a function that maps each v into a value $s(v) \in D_v$; s_* is the *goal*, in the form of a partial state over V; and O is a set of *operators*, mapping between states. For convenience, we use some specialized notations for states. A *fact* is an assignment $v = d$. The set of all facts is denoted with P. We often interchange variable values with facts, and we treat (partial) states as sets of facts. For a fact p, by $var(p)$ and $val(p)$ we denote the variable and value associated with p, respectively. By $s(p)$ we denote the fact $q \in s$ where $var(p) = var(q)$ (intuitively, "the value of $var(p)$ in s"). By $s[p]$ we denote the state that is like s except that $var(p)$ is assigned to $val(p)$.

Our notation for operators is slightly unusual, but convenient for our purposes herein. In terms of PDDL terminology, the operators are obtained by moving the action precondition into the effect conditions, and by generating a separate effect for each effect literal. Hence, each operator $o \in O$ consists of a set of *rules* $r : Z_r \rightarrow x_r$. In each rule, the *effect* x_r is a fact, the *condition* Z_r is a set of facts, and there is exactly one *pivot condition* $x''_r \in Z_r$ where $var(x''_r) = var(x_r)$. The latter assumption is not limiting because, if x''_r is not present in Z_r, then the rule can be replaced with a set of rules, one for each value in $D_{var(x_r)}$. We denote the set of all rules (of all $o \in O$ together) with R.

The application of an operator o in a state yields a new state s' that is like s except that variable v is assigned value d whenever $r : Z_r \rightarrow v = d$ is a rule of o and $Z_r \subseteq s$. We assume that operators are not self-contradictory, i.e., that s' is always well-defined. A *plan* is a sequence of operators that maps the initial state into a final state s_G with $s_* \subseteq s_G$.

Context-Enhanced Additive Heuristic

As mentioned, our approach extends the context-enhanced additive heuristic h^{cea}, which in turn extends the additive heuristic h^{add}. We introduce both h^{add} and h^{cea}, in a slightly changed notation which is useful to establish more directly the connection to our extended heuristic h^{pcc}.

We set $h^{add}(s) := \Sigma_{x \in s^*} h^{add}(x|s)$ where $h^{add}(x|s') :=$

$$\begin{cases} 0 & \text{if } x \in s' \\ 1 + \min_{r:x_r=x}[\Sigma_{y \in Z_r} h^{add}(y|s')] & \text{if } x \notin s' \end{cases} \quad (1)$$

This definition of h^{add} anticipates the use of a context state s' relative to which the cost value of the conditions $y \in Z_r$ is computed. Since s' is never modified in the recursion, it is always equal to the actual state s. This is not so for h^{cea}, defined by $h^{cea}(s) := \Sigma_{x \in s^*} h^{cea}(x|s)$ where $h^{cea}(x|s') :=$

$$\begin{cases} 0 & \text{if } x \in s' \\ 1 + \min_{r:x_r=x}[h^{cea}(x''_r|s') + & \text{if } x \notin s' \\ \Sigma_{y \in Z_r \setminus \{x''_r\}} h^{cea}(y|s[s^{cea}(x''_r|s')(y)])] & \end{cases} \quad (2)$$

In this definition, the cost value for x''_r – the pivot condition – is evaluated relative to the same context state s'. However, the other conditions $y \in Z_r \setminus \{x''_r\}$ are evaluated relative to *the state that results from achieving x''_r in s'*. More precisely, $s^{cea}(x''_r|s')$ (defined below) is an approximation of that state. To ensure that only a small number of different context states is generated, $s^{cea}(x''_r|s')$ is *projected onto* $var(y)$: it is only considered how the context differs from s with respect to $var(y)$. Note that projection is an additional approximation. Using it, every context state ever considered differs from s in at most one variable value. Precisely, in every recursive instance of $h^{cea}(x|s')$ and $s^{cea}(x|s')$ either $s' = s$ or s' differs from s at most in the value of $var(x)$.[1]

It remains to define $s^{cea}(x|s')$, i.e., to specify the approximation of the state that arises when achieving a fact x from a context state s'. We set $s^{cea}(x|s') :=$

$$\begin{cases} s' & \text{if } x \in s' \\ s^{cea}(x''_{r^m}|s')[r^m] & \text{if } x \notin s' \end{cases} \quad (3)$$

where r^m is the rule that yields the minimum in Eq. 2, and $s[r^m]$ denotes s modified by, in this order, Z_{r^m}, x_{r^m}, and the set of "side effects" of r^m, i.e., the effects of those rules r' of the same operator where $Z_{r'} \subseteq Z_{r^m}$. In words, we recurse to approximate the state resulting from achieving the pivot of the "best" rule r^m as per Eq. 2, and we apply to that state an approximation of the changes made by r^m itself.

Precedence Constraints Contexts Heuristic

We define the h^{pcc} heuristic function, extending h^{cea} to handle arbitrary precedence constraints. We first give an important notation, then we introduce the extended equations, then we discuss some alternative definitions. We analyze the relation to h^{add} and h^{cea}.

Context Functions

We will consider in the next section how to generate precedence constraints. More precisely, we will define strategies that construct, in a pre-process, a *context function*. This is a partial function $ctx : R \times P \mapsto P$, where $ctx(r, q) = p$ indicates that the context of condition $q \in Z_r$ should be the state that results from achieving $p \in Z_r$.

For defining h^{pcc}, it does not matter how the context function is constructed. Our only assumption is that the function does not have cycles in the corresponding graph of precedence constraints, i.e., the graph with nodes Z_r and edges $\{(p, q) \mid ctx(r, q) = p\}$. Each node in this graph has at most one parent (ctx is a function), so this graph is a forest. We denote with $Z^L_r := \{y \mid y \in Z_r, \text{not ex. } y' \in Z_r \text{ s.t. } ctx(r, y') = y\}$ the leaves of that forest, and with $Z^R_r := \{y \mid y \in Z_r, ctx(r, y) \text{ is undef.}\}$ its roots.

The ctx function is called *trivial* if it is not defined for any r and q; ctx is called *pivot-based* if, for all $r \in R$, $ctx(r, x''_r)$ is undefined and, for all $y \in Z_r \setminus \{x''_r\}$, $ctx(r, y) = x''_r$. We will see that, for trivial respectively pivot-based ctx functions, h^{pcc} coincides with h^{add} respectively with h^{cea}.

Definition of h^{pcc}

The overall heuristic function h^{pcc} is defined by $h^{pcc}(s) := \Sigma_{x \in s^*} h^{pcc}(x|s)$, with $h^{pcc}(x|s') :=$

$$\begin{cases} 0 & \text{if } x \in s' \\ 1 + \min_{r:x_r=x}[\Sigma_{y \in Z_r} h^{pcc}(y|c^{pcc}(s', r, y))] & \text{if } x \notin s' \end{cases} \quad (4)$$

[1] Accordingly, Helmert and Geffner (2008) define equations $h^{cea}(x|x')$ and $s^{cea}(x|x')$ instead, where $var(x) = var(x')$. Our definition is more explicit about the use of projection.

The structure of this equation corresponds in a straightforward way to Eq. 2. The only difference is that we do not explicitly distinguish between the context states for the pivot condition x''_r vs. all other conditions. Indeed the equation does not immediately give any hint on how the context states are defined, leaving this to the function $c^{pcc}(s', r, y)$. That function defines *the context state from which y should be achieved, given that r is applied to achieve x from context state s'*. This new context state does not depend on x (the effect of r is not relevant to achieving its conditions), which is why the function takes only s', r, and y as its arguments. Formally, we define $c^{pcc}(s', r, y) :=$

$$\begin{cases} s[s'(y)] & \text{if } ctx(r, y) \text{ is undef.} \\ s[s^{pcc}(p|c^{pcc}(s', r, p))(y)] & \text{if } ctx(r, y) = p \end{cases} \quad (5)$$

Notice first that, as for h^{cea}, the context state is projected onto y, ensuring a small number of different context states.

If $ctx(r, y)$ is undefined – the first case of Eq. 5 – then the context state for y should be the same as the context state for r, namely s'. If, on the other hand, $ctx(r, y) = p$ then the context state for y should be the state that results from achieving p, in its respective context state. The context state for p is captured by recursing to $c^{pcc}(s', r, p)$.[2] The role of the s^{pcc} function is similar to the role of s^{cea} for h^{cea}. The way it is used here, it defines an approximation of *the state that results from achieving p in its context state.*

More generally, s^{pcc} has as arguments some fact x and context state s'. Recall that the context function can be understood as a forest over Z_r, whose leaves are denoted with Z^L_r. We define $s^{pcc}(x|s') :=$

$$\begin{cases} s' & \text{if } x \in s' \\ s^{pcc}(y_0|c^{pcc}(s', r^m, y_0))[r^m], \text{ where } y_0 \in Z^L_{r^m} & \text{if } x \notin s' \end{cases} \quad (6)$$

with r^m being the rule that yields the minimum in Eq. 4. As for the first case, if x is already true in s' then there is nothing to do (x is already achieved in its context). In the other case, we consider the "best" rule r^m for achieving x, according to Eq. 4. What is the outcome state of applying this rule? We need to (1) achieve its conditions in the order dictated by ctx (by the precedence constraints), and then (2) apply the changes made by r^m itself. (1) is captured by $s^{pcc}(y_0|c^{pcc}(s', r^m, y_0))$: this is the state that results from achieving a leaf condition in its respective context state. Since y_0 is a leaf, it is achieved "last". Of course, there may be several leaves, with no order enforced by the precedence constraints. We choose an arbitrary one.[3] (2) is then captured in the same way as for h^{cea}.

Example 1 *Reconsider the Grid example mentioned in the introduction. We have $r : locked(D), at(L), have(K) \rightarrow unlocked(D)$. In the current context state s', the key is at $L' \neq L$. We wish our heuristic to account for the cost of moving from L' to L. This should be accomplished by achieving first $have(K)$, yielding a new context state in which we then achieve $at(L)$. We can reflect this by setting $ctx(r, at(L)) = have(K)$. Consider the cost estimate for $at(L)$. By Eq. 4, this is $h^{pcc}(at(L)|c^{pcc}(s', r, at(L)))$. By Eq. 5, $c^{pcc}(s', r, at(L))$ arises from achieving $have(K)$ in $c^{pcc}(s', r, have(K))$. Clearly, the best rule r^m here is the rule that picks K up at L', so according to Eq. 6, which enforces the conditions of r^m, we get $at(L') \in s^{pcc}(have(K)|c^{pcc}(s', r, have(K)))$. So $c^{pcc}(s', r, at(L)) = s[at(L')]$; and $h^{pcc}(at(L)|s[at(L')])$ yields the cost for moving from L' to L.*

Note here that the ability to account for the cost for moving from L' to L is quite remarkable, and well beyond the capabilities of, e.g., the relaxed plan heuristic.

Alternative Definitions

We have given above the most straightforward and intuitive versions of the extended equations. Having a closer look, one finds that there are several noteworthy alternatives.

First, note in Eq. 6 the following departure from h^{cea}. In Eq. 3, the effect of r^m is applied to the state in which the pivot condition x''_r was achieved. In our terms, this is a *root* condition, rather than a leaf as taken by Eq. 6 – h^{cea} orders the pivot before all other conditions (which is formally reflected in pivot-based context functions).

Computing s^{pcc} based on a root condition does not make much sense intuitively. Presumably this point was overlooked in h^{cea}, where the "context functions" are very simple. Still, this being the configuration of our direct predecessor heuristic, we consider it as an option:

$$\text{use } y_0 \in Z^R_{r^m} \text{ in Eq. 6} \quad (6B)$$

For h^{pcc} to "strictly" generalize h^{cea}, the two heuristics should coincide whenever the ctx function is pivot-based. Theorem 2 below shows that this holds true for h^{pcc} based on Eq. 6B. It is *not* true for Eq. 6:

Proposition 1 *There exist a task (V, s_0, s_*, O) with pivot-based context function ctx, and a state s, so that $h^{pcc}(s)$ based on Eq. 6 is different from $h^{pcc}(s)$ based on Eq. 6B.*

Proof Sketch: It is easy to construct an example where achieving a leaf condition has a side effect overlooked when computing s^{pcc} according to Eq. 6B. ∎

Hence h^{pcc} based on Eq. 6 does not strictly generalize h^{cea}. We will show later that the empirical difference between Eqs. 6 and 6B is moderate, with a slight advantage for Eq. 6.

Our next observation may come as somewhat of a shock. It may happen that the heuristic cost of the pivot element x''_r, relative to the original context state s', is higher than the sum of the heuristic costs of *all conditions y together* (including x''_r itself), relative to their actual context states $c^{pcc}(s', r, y)$:

Proposition 2 *There exist a task (V, s_0, s_*, O), context function ctx, and state s, so that recursion on $h^{pcc}(s)$ results in an instance of $h^{pcc}(x|s')$ where $h^{pcc}(x''_r|s') > \Sigma_{y \in Z_r} h^{pcc}(y|c^{pcc}(s', r, y))$.*

Proof Sketch: We construct an example where x''_r is much harder to achieve from s' than from s, in particular $s(x) \neq s'(x)$. The construction has $ctx(r, x''_r) = y$, i.e., the context of x''_r, $c^{pcc}(s', r, x''_r)$, is computed from $c^{pcc}(s', r, y)$. In the latter, s' is projected onto $var(y)$, losing the distinction between $s(x)$ and $s'(x)$. We get $c^{pcc}(s', r, x''_r)(x) = s(x)$, from which the claim follows. ∎

[2] This recursion terminates because ctx does not have cycles.

[3] Since there are several other and more important design decisions (to be discussed shortly), we did not yet experiment with alternative solutions to this issue.

In other words, due to the approximations that we inherit from h^{cea} – projection, specifically – it may yield a better estimate to simply achieve the pivot in s', rather than to achieve the whole condition in the contexts derived from the precedence constraints! Clearly, such behavior is pathological. A simple remedy is:

$$\max \text{ over given expression and } h^{pcc}(x''_r|s') \text{ in Eq. 4} \quad (4B)$$

Here we insert the pivot as a kind of fallback option. Alternatively, we can set:

$$\max \text{ over given expression and } h^{pcc}(x''_r|s') + \Sigma_{y \in Z_r \setminus \{x''_r\}} h^{pcc}(y|c^{pcc}(s', r, y)) \text{ in Eq. 4} \quad (4C)$$

This alternative "tries the pivot both ways", $\max$'ing over its use with context state s' vs. context state $c^{pcc}(s', r, x''_r)$.

All three alternatives have merits. Eq. 4 is most natural. Eq. 4B is the least intrusive fix for the pathological behavior identified by Proposition 2. Eq. 4C makes use of additional information (the cost of the other conditions) ignored by Eq. 4B. In our experiments, we try all these options.

Relation to h^{add} and h^{cea}

We build on the following observation: *(*) in any recursive instance of $h^{pcc}(x|s')$ and $s^{pcc}(x|s')$, s' is either identical to s or differs from s only in $var(x)$.* This can easily be seen by induction on the recursion structure. We get:

Theorem 1 *Let (V, s_0, s_*, O) be a task, and ctx be the trivial context function. Then, regardless whether Eq. 4 or 4B or 4C is used, and regardless whether Eq. 6 or 6B is used, h^{pcc} coincides with h^{add}.*

Proof Sketch: Given that $ctx(r, y)$ is undefined for all y, we can insert the first case of Eq. 5 into the second case of Eq. 4, obtaining the expression $\Sigma_{y \in Z_r} h^{pcc}(y|s[s'(y)])$. With (*), for $y = x''_r$ the context state $s[s'(y)]$ simplifies to s', and for all other y that state simplifies to s. The resulting equation is obviously equivalent to Eq. 1. Eq. 6 and 6B are irrelevant to this argument. As for Eq. 4B and 4C, the contexts on both sides of the maximization are the same. ∎

Theorem 2 *Let (V, s_0, s_*, O) be a task, and ctx be the pivot-based context function. Then, regardless whether Eq. 4 or 4B or 4C is used, h^{pcc} based on Eq. 6B coincides with h^{cea}, provided the tie breaking for the choice of the rules r^m is identical.*

Proof Sketch: We simplify the definitions of h^{pcc} and s^{pcc} so that they coincide with those of h^{cea} and s^{cea}. Consider first Eq. 4. With (*) and since $ctx(r, x''_r)$ is undefined, the context state for x''_r is s', like in Eq. 2. For all other y, since $ctx(r, y) = x''_r$, by Eq. 5 and (*) we get $c^{pcc}(s', r, y) = s[s^{pcc}(x''_r|c^{pcc}(s', r, x''_r))(y)] = s[s^{pcc}(x''_r|s')(y)]$. As desired, this is identical (modulo function names) to the context $s[s^{cea}(x''_r|s')(y)]$ used in Eq. 2. Consider now Eq. 6B, returning $s^{pcc}(y_0|c^{pcc}(s', r^m, y_0))[r^m]$ where $y_0 \in Z^R_{r^m}$. By prerequisite, $Z^R_{r^m} = \{x''_{r^m}\}$ so we obtain $s^{pcc}(x''_{r^m}|c^{pcc}(s', r^m, x''_{r^m}))[r^m]$. Since $ctx(r, x''_{r^m})$ is undefined, and with (*), like above this simplifies to $s^{pcc}(x''_{r^m}|s')[r^m]$. The latter corresponds exactly to $s^{cea}(x''_{r^m}|s')[r^m]$ because, by prerequisite, ties in the choice of r^m (if several r^m yield the minimum) are broken in the same way. As for Eq. 4B and 4C, the contexts for x''_r on both sides of the maximization are the same. ∎

Generating Precedence Constraints

Given a planning task, how should we automatically generate precedence constraints (and thus a context function)? Clearly, the choice of constraints will determine the quality of the heuristic function. But what are "good" constraints? On which basis should we order action conditions? Herein, we pursue a natural answer suggested directly by Helmert and Geffner's (2008) Grid example: we leverage on previous work in the area of goal (and sub-goal) orderings (Koehler and Hoffmann 2000; Hoffmann, Porteous, and Sebastia 2004; Richter, Helmert, and Westphal 2008).

Consider again the rule "locked(D), at(L), have(K) → unlocked(D)", and the desired precedence constraint have(K) < at(L). This is exactly what Koehler and Hoffmann (2000) term a *reasonable* order. Two goals p and q are reasonably ordered $p <_r q$ if achieving p involves, as a side effect, deleting q. In this situation, achieving q first results in wasted effort because, once p is achieved, q must be reestablished. Hence the "reasonable" order. From our perspective here, the order is also entirely appropriate, since achieving p may affect the cost of achieving q, making it more costly. In the Grid example, to achieve have(K) we must delete at(L). Informing the heuristic of this fact enables it, as we have seen in Example 1, to account for the cost of moving from K's current position to L.

Given the above, a natural question to ask is whether we can also leverage on the known generalizations of Koehler and Hoffmann's goal orderings. Hoffmann et al. (2004) and Richter et al. (2008) order *landmarks*, facts that must be true at some point in any plan. Beside (an adaptation of) reasonable orders, such facts can be *necessarily* ordered $p <_n q$ if any plan that achieves q must achieve p beforehand. Note that, in this setting, p is a support'for q, unlike the conflicts underlying reasonable orders: ordering p before q may decrease the cost of q. Since the heuristic is not admissible, such a cost decrease may yield a more accurate estimate.

Although the connections discussed above are intuitive, one should keep in mind that goal orders were developed in a very different context and with very different intentions. A priori we have no idea how they affect performance when used for the generation of context functions. Hence our general rationale is to try every possible configuration, as long as it has at least some (perhaps far-fetched) motivation. Our implementation is based on that of Richter et al., which is recent and features rich techniques for detecting sound necessary orders. We experiment with the following variants:

1. **Aorg:** In this variant we detect reasonable and necessary orders between arbitrary pairs of facts (regardless whether or not they are landmarks or goals). This is the most straightforward technique and matches well the intuitions given above. Note that, in our setting, the contexts concern facts participating in the same operator condition, so that it does not matter whether they are goals/landmarks.
2. **Ainv:** Like **Aorg** except that we *invert* the necessary orders: if the landmarks analysis returns an ordering $p <_n q$ then we set $q < p$. This is motivated by an observation in the Grid example. If L lies on the best path towards K, we get at(L) $<_n$ have(K), the opposite of the order we desire.

3. **Lorg:** Like **Aorg** except that we order only pairs of landmarks, hence using exactly the orderings generated by the code of Richter et al. While there is no reason to emphasize on landmarks, this version is of a generic interest and serves to show whether the distinction between landmarks/no landmarks makes an empirical difference.
4. **Linv:** Like **Lorg** except that necessary orders are inverted as in **Ainv**.
5. **INorg:** Like **Aorg** except that we use only necessary orders $p <_n q$. This serves to test the effect of necessary orders in isolation.
6. **ININinv:** Like **INorg** but inverting the orders.
7. **IR:** Like **INorg** but taking only reasonable orders $p <_r q$, hence testing their effect in isolation.

One of these options is chosen by the user. The orderings are then computed in a pre-process to planning, resulting in a partial order "$<$" over the facts.[4]

To generate the ctx function, we look at each rule r and each $q \in Z_r$ in turn. We construct the set C which contains all $p \in Z_r$ where $p <^* q$ and there does not exist $p' \in Z_r$ with $p <^* p' <^* q$; "$<^*$" here denotes the transitive closure of "$<$". We set $ctx(r, q) := p$ where $p \in C$. If $|C| > 1$, then a choice needs to be made. We choose p arbitrarily, motivated by the observation that this situation rarely occurs: in tests with all seven strategies and 31 planning benchmark domains, $|C| > 1$ happened at all only in about half of the 217 combinations of strategy/domain, and where it did happen, the fraction of such cases was mostly below 5%.

Experiments

We run experiments in all 31 domains from the International Planning Competitions up to 2006 (IPC 1–5), apart from those solved trivially (Movie and Gripper). The experiments were conducted on a heterogeneous cluster of Intel Xeon and AMD Opteron CPUs, ranging from 2.2 GHz to 2.83 GHz. (The magnitude of the experiments precluded experiments on homogeneous machines.) To allow fair comparisons, for each given planning task, all planner configurations were run on the same CPU. The memory limit was 1.75 GB in all cases. All heuristic functions are implemented within Fast Downward (Helmert 2006), and all were run in exactly the same search algorithm, greedy best-first search with deferred evaluation and (sometimes, depending on the experiment) preferred operators. We examine three questions:

1. How does the configuration of h^{pcc} – Eq. 4 vs. Eq. 4B vs. Eq. 4C, and Eq. 6 vs. Eq. 6B – affect performance?
2. How does the variant of precedence constraints – **Lorg** vs. **Linv** vs. **Aorg** vs. **Ainv** vs. **INorg** vs. **ININinv** vs. **IR** – affect performance?
3. How does h^{pcc} perform compared to h^{cea}?

If we vary all parameters in combination, we get $3 * 2 * 7 = 42$ versions of h^{pcc}, which is too much for a comprehensive analysis within the given space. Instead, we separate question 1 from questions 2 and 3. We first examine the effect of different configurations, fixing just 2 possible variants of precedence constraints. Based on the outcome, we fix a best configuration, which we then use for answering questions 2 and 3, where we compare all 7 variants of precedence constraints against h^{cea}. We will see that we can often improve on h^{cea}. However, the extent of the improvement is moderate, and behavior is usually not consistent across the 7 variants. We shed some light on this with an analysis based on sampling from *all possible* sets of precedence constraints.

[4]Reasonable orderings may cause cycles; these are broken by removing the culprit pairs $p <_r q$.

	IR			Linv		
Domain	**Eq. 4**	**Eq. 4B**	**Eq. 4C**	**Eq. 4**	**Eq. 4B**	**Eq. 4C**
Airport	$+2/-1$	$+3/-2$	$+2$	$+5$	$+4$	
Depot		-2				-1
FreeCell	-2		$+1$	$+3/-2$	-1	
Grid				-1	$+1$	
Logistics-1998	-1			-2	-1	-1
Miconic-ADL	$+1$	$+1$				
MPrime		-1				$+1$
Openstacks			$+1$		$+1$	$+12$
Philosophers			$+1$			
Pipesworld-NoTankage		-1				-1
Pipesworld-Tankage	-1	-1	$+1$	-2		
PSR-Large				$+1$		
Trucks	-1	-1	$+1$			
total	-3	-4	$+7$	$+2$	$+4$	$+10$

Table 1: Improvement in coverage when using Eq. 6 rather than Eq. 6B. An entry like "$+2/-1$" means that two instances are solved with Eq. 6 but not with Eq. 6B, and the opposite is true for one instance. For all empty entries and all IPC 1–5 domains not shown, the same set of instances was solved with Eq. 6 and Eq. 6B.

Which configuration of h^{pcc} to use?

We evaluate the effect of using the different variants of our equations. We do not use preferred operators since we are exclusively concerned with the relative behavior of the different variants. We restrict the third parameter of h^{pcc} – the method for generating precedence constraints – arbitrarily to the 2 options **IR** and **Linv**. For each combination of domain and setting of two parameters, we examine what happens as we change the value of the free parameter. Precisely, we consider pairs of possible values x and y of the free parameter, and examine how performance gets better/worse when using x rather than y. We summarize this performance delta in terms of coverage.

Table 1 presents our data for the use of Eq. 6 rather than Eq. 6B. Observe first that this choice affects coverage in less than half of our domains (13 out of 29). For the affected domains, the results are not entirely conclusive, but there is an advantage for Eq. 6, i.e., for defining context states based on tree leaves rather than tree roots. The advantage is most consistent for **IR** and Eq. 4C; overall (see bottom row), Eq. 6 has an advantage in four of the six possible settings of the other parameters. We hence choose Eq. 6 for our experiments in the subsequent section.

For the three different variants of Eq. 4, we need to compare all three pairs of values. First, we consider the pair Eq. 4B vs. Eq. 4, i.e., we consider the advantage of using our straightforward fix for the pathological behavior identified in Proposition 2. This is a story quickly told: in 120 combinations of domain and surrounding parameter settings, there is only a single instance that is solved with Eq. 4 but not with

Domain	IR		Linv	
	Eq. 6	Eq. 6B	Eq. 6	Eq. 6B
Airport	+5/−1	+4/−1	+12/−1	+15
Blocks	+2	+2		
Depot	+2			+1
Driverlog	+2	+2		
FreeCell	+2/−1	+1/−1	+2/−1	+1/−1
Grid			−1	
Miconic-ADL	−1			
MPrime	+1		+1	
Openstacks		−1	+15	+4
Pathways	+1/−2	+1/−2		
Philosophers	−37	−38		
Pipesworld-NoTankage	+1/−1	−1	−2	−1
Pipesworld-Tankage	+2	+1/−1	+1/−2	+1/−2
PSR-Small	+1	+1		
Rovers	+3/−1	+3/−1	−1	−1
Storage			−1	−1
Trucks	+2	+1/−1		
total	−20	−31	+22	+16

Table 2: Improvement in coverage when using Eq. 4B rather than Eq. 4C (notation as in Table 1).

Eq. 4B, namely in Pipesworld-NoTankage when using **IR** and Eq. 6; but even there, Eq. 4B is better overall, solving two instances not solved with Eq. 4.

Somewhat surprisingly perhaps, Eq. 4C – our slightly more involved fix to Eq. 4 – does not fare nearly as well compared to Eq. 4. In 15 combinations of domain and surrounding parameter settings, a total of 63 instances is solved with Eq. 4 but not with Eq. 4C. This clearly indicates that Eq. 4B is the more reliable fix.

Table 2 compares the two fixes to Eq. 4 directly and in detail. Note that this parameter setting has more impact, with 17 affected domains rather than the 13 for Table 1. For **IR**, Eq. 4C has a big advantage overall (bottom row). Note, however, that this is almost exclusively due to a single domain, Philosophers, which has a rather particular structure. Counting the number of individual domains where one or the other method is in the advantage, we get 10 vs. 3 domains for Eq. 4B with **IR** and Eq. 6, and 6 vs. 4 domains for Eq. 4B with **IR** and Eq. 6B. Focussing on **Linv**, we get significant overall advantages for Eq. 4B, based on several domains rather than just a single one. That said, the number of "winning" domains is almost equal, with 4 vs. 4 in **Linv** with Eq. 4 and 3 vs. 4 in **Linv** with Eq. 4B.

Our choice for the subsequent experiments is to go with Eq. 4B, motivated in part by its slight superiority over Eq. 4C as per Table 2, and motivated more strongly by its much more reliable behavior vs. Eq. 4, as outlined above.

How does h^{pcc} perform compared to h^{cea}?

We now fix the configuration to use Eqs. 4B and 6. We examine the performance of the 7 variants for generating precedence constraints. Since preferred operators yield a huge improvement for overall performance (for any of the heuristics tested), we switch them on.

We provide at the end of this subsection a brief summary of a direct comparison between h^{pcc} and h^{cea}. Our more detailed results compare against what we call h^{sim}: the heuristic function that results from feeding pivot-based ctx functions into h^{pcc} using Eq. 6B. The functions h^{cea} and h^{sim} are, in principle, identical. However, due to practical differences, for an accurate assessment of how h^{pcc} fares compared to h^{cea}, it is more sensible to compare to h^{sim}.

First, Theorem 2 is conditioned on having identical tie breaking for choosing the rules r^m, which is not the case in our independent implementation of h^{pcc}. Hence the search spaces incurred by h^{cea} and h^{sim} may differ. We measured, for every instance solved by both h^{cea} and h^{sim}, what we call the *expansions improvement* for h^{cea} vs. h^{sim}: the number of states expanded with h^{cea}, divided by the number of states expanded with h^{sim}. We take the geometric mean over all instances of each domain. The results range between 0.356 (Driverlog, h^{cea} in the advantage) and 1.486 (Trucks, h^{sim} in the advantage). The geometric mean over all domains is 0.963, indicating a slight advantage for h^{cea}.

Domain	Lorg	Linv	Aorg	Ainv	INorg	INinv	IR	BestOf
Airport	+5/−4	+10	−5	+6	+1/−5	+10	+6/−1	+14
Assembly	+1	+1	+1	+1	+1	+1	+1	+1
Depots	+1/−1	+1	+1/−2	+1/−1	+1/−2	+1/−1	+2/−1	+3
FreeCell	+2	+3/−2	+2	+2/−1	+2	+3	+2	+4
Mystery	−3	−3	−3	−3	−3	−3	−3	−3
Optical-T	−1	−1	−1	−1	−1	−1	−1	−1
Pathways	+1	+1	+1	+1/−1	+1	+1	+1/−1	+1
Philos.			−37	−32		−32	−38	
Pipe-NoT	+3/−4	+3/−6	+3/−2	+4/−2	+3/−3	+4/−5	+3/−4	+4/−1
Pipe-T	+6/−1	+6/−1	+8/−1	+10/−2	+6	+7/−2	+5/−2	+11/−1
PSR-L			−1					
Rovers						−1		
Schedule	+1	+1	+7/−1	−1	−1	+1/−1	−1	+8
Storage	−1		−1	−1	−1	−1	−1	
TPP	+6	+6	+7	+8	+7	+8	+6	+8
Trucks	+3	+2		+4		+2	+2	+6
total	+14	+21	−24	−8	+6	−9	−25	+54

Table 3: Improvement in instances solved when using variants of h^{pcc} rather than h^{sim} (notation as in Table 1). **BestOf** returns, for each instance, the best result obtained with any of the variants for generating precedence constraints.

Second, node expansion is around 5 times faster for h^{cea} than for h^{sim}. This is very likely to be only due to a lack of optimization: the implementation of h^{cea} is quite sophisticated, and in principle there is no reason why h^{sim} should be slower than h^{cea}. The only runtime overhead of h^{pcc} should result from the use of more complex ctx functions.[5] Comparing to h^{sim} allows to account for exactly that.

Table 3 shows, in analogy to Tables 1 and 2, the coverage improvement one gets from using a variant of h^{pcc} rather than h^{sim}. Table 4 complements this in terms of the average expansions improvement, showing the geometric mean for each domain and variant. We provide data also for a hypothetical **BestOf** version that returns, *for each instance*, the best result obtained with any of the h^{pcc} variants. This shows the potential of h^{pcc} when abstracting from the need to choose one of our 7 variants of precedence constraints. It also nicely illustrates how very complementary these variants are, with greatly varying benefits even for individual instances within the same domain.

From a quick glance at Tables 3 and 4, one can draw the following main conclusions:

- *The choice of precedence constraints is a critical one*, with particular variants yielding advantages in some cases but disadvantages in others.
- *The scale of the effect on performance is generally not dramatic*, below or within one order of magnitude.
- *In about half of the domains, h^{pcc} can bring significant improvements.*

[5] Note that the maximization used in Eqs. 4B and 4C may contribute to such an overhead; but not for pivot-based ctx functions as in h^{sim}, where both sides of the maximization are the same.

Domain	Lorg	Linv	Aorg	Ainv	INorg	ININv	IR	BestOf
Airport	0.706	1.360	0.716	1.187	0.743	1.028	0.879	2.182
Assembly	0.835	0.835	1.082	0.850	1.082	0.851	0.835	1.268
Blocks	0.996	1.141	0.997	1.154	0.996	1.049	1.127	1.374
Depots	0.882	2.048	1.499	1.194	0.787	1.633	1.042	2.624
Driverlog	1.197	1.197	1.100	1.100	1.245	1.114	1.090	1.269
FreeCell	0.946	1.007	1.004	1.000	0.948	0.985	0.928	1.455
Grid	0.994	0.987	1.025	2.521	1.513	1.212	2.189	2.647
Log	0.802	0.858	0.805	0.858	0.800	0.812	0.858	0.890
Mic-ADL	0.999	0.996	1.061	1.045	1.000	1.003	1.082	1.186
Mic-Sim	1.209	1.198	1.602	3.554	1.018	1.126	3.554	3.554
Mic-STR	2.591	2.843	1.451	2.843	1.083	1.118	2.843	2.846
MPrime	0.818	0.818	0.813	0.818	0.795	0.795	0.818	0.818
Mystery	0.871	0.871	0.860	0.871	0.826	0.826	0.871	0.871
Openst	1.028	1.029	1.036	1.030	1.017	1.030	1.057	1.063
Optical-T	0.596	0.596	0.662	0.434	0.688	0.600	0.358	0.688
Pathways	1.042	1.042	1.042	1.000	1.042	0.999	1.044	1.060
Philos.	0.722	0.722	0.054	0.133	0.671	0.146	0.041	0.759
Pipe-NoT	0.856	0.647	0.737	0.960	0.853	0.734	0.745	1.807
Pipe-T	0.752	0.846	0.723	0.839	0.899	0.688	0.789	1.770
PSR-L	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.001
PSR-M	1.008	1.000	1.000	1.000	1.000	1.000	1.000	1.010
PSR-S	1.007	0.994	1.012	1.003	1.016	1.005	1.020	1.043
Rovers	1.000	0.992	0.878	0.991	0.994	0.979	1.004	1.216
Satellite	1.028	1.024	0.920	1.146	1.036	1.011	1.225	1.272
Schedule	0.956	1.052	1.942	1.044	0.993	1.035	1.026	2.262
Storage	0.882	0.699	0.929	0.785	0.909	0.781	0.958	1.025
TPP	2.742	2.815	2.814	2.519	2.702	2.707	2.905	3.381
Trucks	1.160	1.161	0.785	2.526	0.674	1.686	1.564	3.634
Zenotravel	0.848	0.848	0.848	0.848	0.848	0.848	0.848	0.848

Table 4: Average expansions improvement, i.e., number of expansions for h^{sim} divided by number of expansions for h^{pcc}, per domain and variant of h^{pcc}.

- *Except in the rather exceptional Philosophers domain, total coverage increases for all variants* (**Lorg** $+14$, **Linv** $+21$, **Aorg** $+13$, **Ainv** $+24$, **INorg** $+6$, **ININv** $+23$, **IR** $+13$).
- *The relative strength of the variants depends considerably on individual instances even within domains.* **BestOf** has better coverage resp. better average expansion than that of any of the single variants in 7 resp. 24 of the 29 domains.

A rough grouping of the domains is as follows:

No major effect. In Miconic-ADL, Openstacks, all versions of PSR, and Rovers, h^{pcc} vs. h^{sim} does not make much of a difference (to varying extents).

Much worse. In Philosophers and Optical-Telegraph, h^{pcc} fares much worse. Note that these are domains of a rather specialized structure, encoding automata transition rules into life-cycles of planning actions.

Slightly worse. In Logistics, MPrime, Mystery, Storage, and Zenotravel, h^{pcc} fares slightly worse. In the 4 transportation variants, the scale of the disadvantage is almost constant across variants and even across domains. This is an interesting contrast to Miconic, whose transportation structure is quite similar. We get back to this shortly.

Slightly better. In Assembly, Blocksworld, Driverlog, FreeCell, Pathways, and Satellite, h^{pcc} fares slightly better.

Much better. In Grid, Miconic-Simple, Miconic-STRIPS, Pipesworld-Tankage, TPP, and Trucks, h^{pcc} fares much better (in Trucks, 2 variants are worse in average expansion, but this does not yield lower coverage whereas all other variants increase coverage). For our motivating example Grid, the improvement is somewhat expected. For the Miconic domains, an explanation is the presence of reasonable orders boarded(p) $<_r$ lift-at(f). This is a useful ordering of conditions for actions releasing a passenger p to her destination floor f, accounting for the cost of moving to f from p's origin floor. In other transportation domains (c.f. above) these orders are less likely to be found because transportables may be picked up from several locations (not only their origin).

Mixed. In Airport, Depots, Pipesworld-NoTankage, and Schedule, significant gains or losses are possible depending on h^{pcc} variant and individual instance.

In Assembly, FreeCell, and the Pipesworld domains, sometimes coverage improvement looks better than that of average expansions. This is partly due to a few instances solved by everyone where h^{sim} has the smallest search space. Another factor is heuristic function runtime. We measured the ratio between expanded nodes per second for h^{sim}, divided by that number for each variant; we took the median from every domain, ignoring trivial instances. The geometric mean across variants is 1.055 for Assembly (indicating a slight slowdown), but 0.583 for FreeCell and 0.214 for the two Pipesworlds, indicating a significant speed-up (the reason for the latter is not clear to us, at the time of writing).

Comparing directly to h^{cea}, we can solve 13 instances more. **BestOf** has better coverage in Depots, FreeCell, Pathways, both Pipesworlds, Schedule, Storage, TPP, and Trucks; h^{cea} has better coverage in Miconic-ADL, Mystery, Optical-Telegraph, and Satellite. Recall, however, that **BestOf** chooses the configuration on a per-instance basis. This hypothetical ability plays a big role here. No single variant has better coverage than h^{cea}. Of the individual domains, the ones where improvements are achieved are: Pathways (all variants); Schedule (**Aorg**); Storage (**Linv**); TPP (all variants); and Trucks (**Ainv**).

Can we do better than this?

The primary purpose of this research is to improve on h^{cea} search space size through more complex ctx functions. That has been achieved, but only to a relatively moderate extent. *Is this due to the limitations of the approach, or due to our particular instantiation?* Could we do better by designing different methods for generating precedence constraints? Towards answering this question, we sample from the space of all possible sets of precedence constraints, for some fixed benchmark instances. From each of our domains we selected 1 instance and ran h^{pcc} 100000 times using in each run a randomly generated set of precedence constraints. For 13 of the domains, the results were interesting (feasible in runtime but not trivial in number of expansions). In only 2 of these 13 domains did some of the 100000 samples yield considerably better performance than the best among our variants of h^{pcc}. This is no proof, but gives some indication that we are close to the limitations of the approach.

It is interesting to consider the results in more detail; see Table 5. Our instances from Airport, Blocksworld, Pipesworld-Tankage, and Rovers all behave similarly to the plots shown in Table 5 (a),(b),(c): the range of performance is large, and **BestOf** is among the best runs while h^{add} and h^{cea} are more in the middle. In Philosophers and TPP, the picture is similar except that h^{cea} (which after all is also a variant of h^{pcc}) does better than the more complex versions.

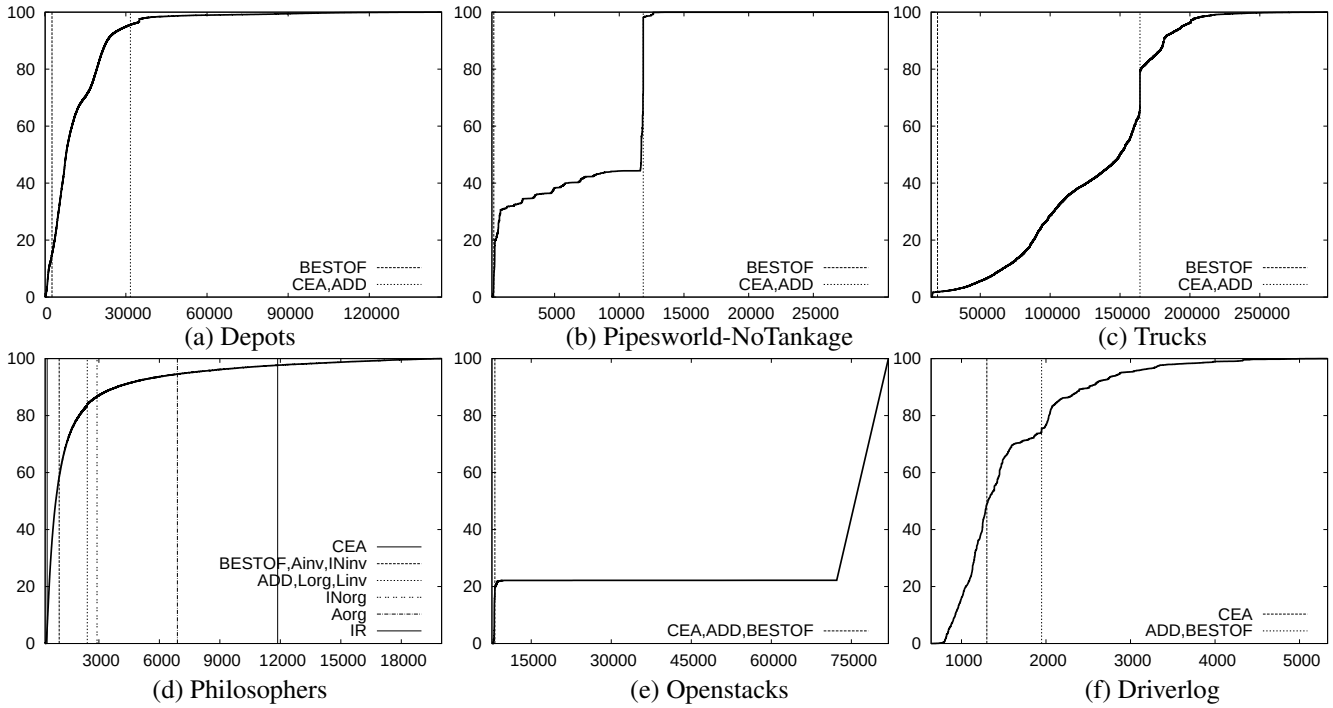

Table 5: Profiling the behavior of all possible sets of precedence constraints. X-axes show number of state expansions, Y-axes show percentage of random runs with $\leq X$ expansions. Vertical lines indicate performance of heuristic functions. In all plots, the lines correspond left-to-right to the keys top-to-bottom (in (b), (d), and (e) the leftmost line is very close to the Y-axis).

In Openstacks, there is a large range of performance but all h^{pcc} variants do perfectly. In Storage, the range of performance is insignificant, i.e., all 100000 samples are close together. Driverlog and Assembly, finally, are the 2 domains where there is considerable room for improvement.

We have in the meantime re-designed the experiment, running 5 different instances from each domain. At the time of writing, only 5000 runs per instance were finished. From this preliminary data, we can see that there often is a lot of variance across instances within a domain. For example, in the new Pipesworld-NoTankage instances **BestOf** is superior to h^{cea} half the time, and vice versa on the other half. Apart from this, the results largely confirm our observations above, with **BestOf** being in the lead in many domains, and a large fraction of random samples doing several orders of magnitude worse than any of the heuristics. The latter indicates that there is significant potential for any modified methods to deteriorate performance – more so, it seems, than for improving it.

Discussion

We answer the question how to extend h^{cea} to general precedence constraints, and how this performs when leveraging on goal ordering techniques. We obtain moderate improvements in a number of benchmarks. We have given some indication that improving this based on different techniques for generating precedence constraints will be difficult.

In our view, the most exciting line of future research is trying to obtain an understanding of why particular kinds of orders yield good/bad performance in particular domains. One option might be to characterize the common properties of random orders underlying particular regions of the plots shown in Table 5.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). For more information, see `http://www.avacs.org/`.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1–2):5–33.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *Proc. ICAPS'08*.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS'04*.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

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

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *JAIR* 22:215–278.

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *JAIR* 12:338–386.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. AAAI'08*.