Speeding Up the Calculation of Heuristics for Heuristic Search-Based Planning

Yaxin Liu, Sven Koenig and **David Furcy**
College of Computing
Georgia Institute of Technology
Atlanta, GA 30312-0280
{yxliu,skoenig,dfurcy}@cc.gatech.edu

Abstract

Heuristic search-based planners, such as HSP 2.0, solve STRIPS-style planning problems efficiently but spend about eighty percent of their planning time on calculating the heuristic values. In this paper, we systematically evaluate alternative methods for calculating the heuristic values for HSP 2.0 and demonstrate that the resulting planning times differ substantially. HSP 2.0 calculates each heuristic value by solving a relaxed planning problem with a dynamic programming method similar to value iteration. We identify two different approaches for speeding up the calculation of heuristic values, namely to order the value updates and to reuse information from the calculation of previous heuristic values. We then show how these two approaches can be combined, resulting in our PINCH method. PINCH outperforms both of the other approaches individually as well as the methods used by HSP 1.0 and HSP 2.0 for most of the large planning problems tested. In fact, it speeds up the planning time of HSP 2.0 by up to eighty percent in several domains and, in general, the amount of savings grows with the size of the domains, allowing HSP 2.0 to solve larger planning problems than was possible before in the same amount of time and without changing its overall operation.

Introduction

Heuristic search-based planners were introduced by (McDermott 1996) and (Bonet, Loerincs, & Geffner 1997) and are now very popular. Several of them entered the second planning competition at AIPS-2000, including HSP 2.0 (Bonet & Geffner 2001a), FF (Hoffmann & Nebel 2001a), GRT (Refanidis & Vlahavas 2001), and AltAlt (Nguyen, Kambhampati, & Nigenda 2002). Heuristic search-based planners perform a heuristic forward or backward search in the space of world states to find a path from the start state to a goal state. In this paper, we study HSP 2.0, a prominent heuristic search-based planner that won one of four honorable mentions for overall exceptional performance at the AIPS-2000 planning competition. It was one of the first planners that demonstrated how one can obtain informed heuristic values for STRIPS-style planning problems to make planning tractable. In its default configuration, it uses weighted A* searches with inadmissible heuristic values to perform forward searches in the space of world states. However, the calculation of heuristic values is time-consuming since HSP 2.0 calculates the heuristic value of each state that it encounters during the search by solving a relaxed planning problem. Consequently, its calculation of heuristic values comprises about 80 percent of its planning time (Bonet & Geffner 2001b). This suggests that one might be able to speed up its planning time by speeding up its calculation of heuristic values. Some heuristic search-based planners, for example, remove irrelevant operators before calculating the heuristic values (Hoffmann & Nebel 2001b) and other planners cache information obtained in a preprocessing phase to simplify the calculation of all heuristic values for a given planning problem (Bonet & Geffner 1999; Refanidis & Vlahavas 2001; Edelkamp 2001; Nguyen, Kambhampati, & Nigenda 2002).

Different from these approaches, we speed up HSP 2.0 without changing its heuristic values or overall operation. In this paper, we study whether different methods for calculating the heuristic values for HSP 2.0 result in different planning times. We systematically evaluate a large number of different methods and demonstrate that, indeed, the resulting planning times differ substantially. HSP 2.0 calculates each heuristic value by solving a relaxed planning problem with a dynamic programming method similar to value iteration. We identify two different approaches for speeding up the calculation of heuristic values, namely to order the value updates and to reuse information from the calculation of previous heuristic values by performing incremental calculations. Each of these approaches can be implemented individually in rather straightforward ways. The question arises, then, how to combine them and whether this is beneficial. Since the approaches cannot be combined easily, we develop a new method. The PINCH (Prioritized, INCremental Heuristics calculation) method exploits the fact that the relaxed planning problems that are used to determine the heuristic values for two different states are similar if the states are similar. Thus, we use a method from algorithm theory to avoid the parts of the plan-construction process that are identical to the previous ones and order the remaining value updates in an efficient way. We demonstrate that PINCH outperforms both of the other approaches individually as well as the methods used by HSP 1.0 and HSP 2.0 for most of the large planning problems tested. In fact, it speeds up the planning time of HSP 2.0 by up to eighty

	Non-Incremental Computations	Incremental Computations
Unordered Updates	VI, HSP2	IVI
Ordered Updates	GBF, HSP1, GD	PINCH

Table 1: Classification of Heuristic-Calculation Methods

percent in several domains and, in general, the amount of savings grows with the size of the domains, allowing HSP 2.0 to solve larger planning problems in the same amount of time than was possible before and without changing its overall operation.

Heuristic Search-Based Planning: HSP 2.0

In this section, we describe how HSP 2.0 operates. We first describe the planning problems that it solves and then how it calculates its heuristic values. We follow the notation and description from (Bonet & Geffner 2001b).

The Problem

HSP 2.0 solves STRIPS-style planning problems with ground operators. Such STRIPS-style planning problems consist of a set of propositions P that are used to describe the states and operators, a set of ground operators O, the start state $I \subseteq P$, and the partially specified goal $G \subseteq P$. Each operator $o \in O$ has a precondition list $Prec(o) \subseteq P$, an add list $Add(o) \subseteq P$, and a delete list $Delete(o) \subseteq P$. The STRIPS planning problem induces a graph search problem that consists of a set of states (vertices) 2^P, a start state I, a set of goal states $\{X \subseteq P | G \subseteq X\}$, a set of actions (directed edges) $\{o \in O | Prec(o) \subseteq s\}$ for each state $s \subseteq P$ where action o transitions from state $s \subseteq P$ to state $s - Delete(o) + Add(o) \subseteq P$ with cost one. All operator sequences (paths) from the start state to any goal state in the graph (plans) are solutions of the STRIPS-style planning problem. The shorter the path, the higher the quality of the solution.

The Method and its Heuristics

In its default configuration, HSP 2.0 performs a forward search in the space of world states using weighted A* (Pearl 1985) with inadmissible heuristic values. It calculates the heuristic value of a given state by solving a relaxed version of the planning problem, where it recursively approximates (by ignoring all delete lists) the cost of achieving each goal proposition individually from the given state and then combines the estimates to obtain the heuristic value of the given state. In the following, we explain the calculation of heuristic values in detail. We use $g_s(p)$ to denote the approximate cost of achieving proposition $p \in P$ from state $s \subseteq P$, and $g_s(o)$ to denote the approximate cost of achieving the preconditions of operator $o \in O$ from state $s \subseteq P$. HSP 2.0 defines these quantities recursively. It defines for all $s \subseteq P$, $p \in P$, and $o \in O$ (the minimum of an empty set is defined to be infinity and an empty sum is defined to be zero):

$$g_s(p) = \begin{cases} 0 & \text{if } p \in s \\ \min_{o \in O | p \in Add(o)} [1 + g_s(o)] & \text{otherwise} \end{cases} \quad (1)$$

$$g_s(o) = \sum_{p \in Prec(o)} g_s(p). \quad (2)$$

```
procedure Main()
  forever do
    set s to the state whose heuristic value needs to get computed next
    for each q ∈ P ∪ O \ s do set x_q := ∞
    for each p ∈ s do set x_p := 0
    repeat
      for each o ∈ O do set x_o := Σ_{p∈Prec(o)} x_p
      for each p ∈ P \ s do set x_p := min_{o∈O|p∈Add(o)} [1 + x_o]
    until the values of all x_q remain unchanged during an iteration
    /* use h_add(s) = Σ_{p∈G} x_p */
```

Figure 1: VI

Then, the heuristic value $h_{add}(s)$ of state $s \in S$ can be calculated as $h_{add}(s) = \sum_{p \in G} g_s(p)$. This allows HSP 2.0 to solve large planning problems, although it is not guaranteed to find shortest paths.

Calculation of Heuristics

In the next sections, we describe different methods that solve Equations 1 and 2. These methods are summarized in Table 1. To describe them, we use a variable p if its values are guaranteed to be propositions, a variable o if its values are guaranteed to be operators, and a variable q if its values can be either propositions or operators. We also use variables x_p (a proposition variable) for $p \in P$ and x_o (an operator variable) for $o \in O$. In principle, the value of variable x_p satisfies $x_p = g_s(p)$ after termination and the value of variable x_o satisfies $x_o = g_s(o)$ after termination, except for some methods that terminate immediately after the variables x_p for $p \in G$ have their correct values because this already allows them to calculate the heuristic value.

VI and HSP2: Simple Calculation of Heuristics

Figure 1 shows a simple dynamic programming method that solves the equations using a form of value iteration. This VI (Value Iteration) method initializes the variables x_p to zero if $p \in s$, and initializes all other variables to infinity. It then repeatedly sweeps over all variables and updates them according to Equations 1 and 2, until no value changes any longer.

HSP 2.0 uses a variant of VI that eliminates the variables x_o by combining Equations 1 and 2. Thus, this HSP2 method performs only the following operation as part of its repeat-until loop:

for each $p \in P \setminus s$ do set $x_p := \min_{o \in O | p \in Add(o)} [1 + \sum_{p' \in Prec(o)} x_{p'}]$.

HSP2 needs more time than VI for each sweep but reduces the number of sweeps. For example, it needs only 5.0 sweeps on average in the Logistics domain, while VI needs 7.9 sweeps.

Speeding Up the Calculation of Heuristics

We investigate two orthogonal approaches for speeding up HSP 2.0's calculation of heuristic values, namely to order the value updates (ordered updates) and to reuse information from the calculation of previous heuristic values (incremental computations), as shown in Table 1. We then describe how to combine them.

```
procedure Main()
  forever do
    set s to the state whose heuristic value needs to get computed next
    for each q ∈ P ∪ O \ s do set x_q := ∞
    for each p ∈ s do set x_p := 0
    repeat
      for each o ∈ O do
        set x_o^new := Σ_{p∈Prec(o)} x_p
        if x_o^new ≠ x_o then
          set x_o := x_o^new
          for each p ∈ Add(o) do
            set x_p := min(x_p, 1 + x_o)
    until the values of all x_q remain unchanged during an iteration
    /* use h_add(s) = Σ_{p∈G} x_p */
```

Figure 2: GBF

IVI: Reusing Results from Previous Searches

When HSP 2.0 calculates a heuristic value $h_{add}(s)$, it solves equations in $g_s(p)$ and $g_s(o)$. When HSP 2.0 then calculates the heuristic value $h_{add}(s')$ of some other state s', it solves equations in $g_{s'}(p)$ and $g_{s'}(o)$. If $g_s(p) = g_{s'}(p)$ for some $p \in P$ and $g_s(o) = g_{s'}(o)$ for some $o \in O$, one could just cache these values during the calculation of $h_{add}(s)$ and then reuse them during the calculation of $h_{add}(s')$. Unfortunately, it is nontrivial to determine which values remain unchanged. We explain later, in the context of our PINCH method, how this can be done. For now, we exploit the fact that the values $g_s(p)$ and $g_{s'}(p)$ for the same $p \in P$ and the values $g_s(o)$ and $g_{s'}(o)$ for the same $o \in O$ are often similar if s and s' are similar. Since HSP 2.0 calculates the heuristic values of all children of a state in a row when it expands the state, it often calculates the heuristic values of similar states in succession. This fact can be exploited by changing VI so that it initializes x_p to zero if $p \in s$ but does not re-initialize the other variables, resulting in the IVI (Incremental Value Iteration) method. IVI repeatedly sweeps over all variables until no value changes any longer. If the number of sweeps becomes larger than a given threshold then IVI terminates the sweeps and simply calls VI to determine the values. IVI needs fewer sweeps than VI if HSP 2.0 calculates the heuristics of similar states in succession because then the values of the corresponding variables x_q tend to be similar as well. For example, IVI needs only 6.2 sweeps on average in the Logistics domain, while VI needs 7.9 sweeps.

GBF, HSP1 and GD: Ordering the Value Updates

VI and IVI sweep over all variables in an arbitrary order. Their number of sweeps can be reduced by ordering the variables appropriately, similarly to the way values are ordered in the Prioritized Sweeping method used for reinforcement learning (Moore & Atkeson 1993).

Figure 2 shows the GBF (Generalized Bellman-Ford) method, that is similar to the variant of the Bellman-Ford method in (Cormen, Leiserson, & Rivest 1990). It orders the value updates of the variables x_p. It sweeps over all variables x_o and updates their values according to Equation 2. If the update changes the value of variable x_o, GBF iterates over the variables x_p for the propositions p in the add list of operator o and updates their values according to Equation 1. Thus, GBF updates the values of the variables x_p only if their values might have changed.

```
procedure Main()
  forever do
    set s to the state whose heuristic value needs to get computed next
    for each q ∈ P ∪ O \ s do set x_q := ∞
    for each p ∈ s do set x_p := 0
    set L_p := s
    while L_p ≠ ∅
      set L_o := the set of operators with no preconditions
      for each p ∈ L_p do set L_o := L_o ∪ {o|p ∈ Prec(o)}
      set L_p := ∅
      for each p ∈ P do set x_p^new := x_p
      for each o ∈ L_o do
        set x_o := Σ_{p∈Prec(o)} x_p
        for each p ∈ Add(o)
          if x_p^new > 1 + x_o
            set x_p^new := 1 + x_o
            set L_p := L_p ∪ {p}
      for each p ∈ P do set x_p := x_p^new
    /* use h_add(s) = Σ_{p∈G} x_p */
```

Figure 3: HSP1

Figure 3 shows the method used by HSP 1.0. The HSP1 method orders the value updates of the variables x_o and x_p, similar to the variant of the Bellman-Ford method in (Bertsekas 2001). It uses an unordered list to remember the propositions whose variables changed their values. It sweeps over all variables x_o that have these propositions as preconditions and updates their values according to Equation 2. After each update of the value of a variable x_o, GBF iterates over the variables x_p for the propositions p in the add list of operator o and updates their values according to Equation 1. The unordered list is then updated to contain all propositions whose variables changed their values in this step. Thus, HSP1 updates the values of the variables x_o and x_p only if their values might have changed.

Finally, the GD (Generalized Dijkstra) method (Knuth 1977) is a generalization of Dijkstra's graph search method (Dijkstra 1959) and uses a priority queue to sweep over the variables x_o and x_p in the order of increasing values. Similar to HSP1, its priority queue contains only those variables x_o and x_p whose values might have changed. To make it even more efficient, we terminate it immediately once the values of all $g_s(p)$ for $p \in G$ are known to be correct. We do not give pseudocode for GD because it is a special case of the PINCH method, that we discuss next.[1]

PINCH: The Best of Both Worlds

The two approaches for speeding up the calculation of heuristic values discussed above are orthogonal. We now describe our main contribution in this paper, namely how to combine them. This is complicated by the fact that the approaches themselves cannot be combined easily because the methods that order the value updates exploit the property that the values of the variables cannot increase during each computation of a heuristic value. Unfortunately, this property no longer holds when reusing the values of the variables from the calculation of previous heuristic values. We

[1] PINCH from Figure 5 reduces to GD if one empties its priority queue and reinitializes the variables x_q and rhs_q before SolveEquations() is called again as well as modifies the while loop of SolveEquations() to terminate immediately once the values of all $g_s(p)$ for $p \in G$ are known to be correct.

thus need to develop a new method based on a method from algorithm theory. The resulting PINCH (Prioritized, INCremental Heuristics calculation) method solves Equations 1 and 2 by calculating only those values $g_{s'}(p)$ and $g_{s'}(o)$ that are different from the corresponding values $g_s(p)$ and $g_s(o)$ from the calculation of previous heuristic values. PINCH also orders the variables so that it updates the value of each variable at most twice. We will demonstrate in the experimental section that PINCH outperforms the other methods in many large planning domains and is not worse in most other large planning domains.

DynamicSWSF-FP

In this section, we describe DynamicSWSF-FP (Ramalingam & Reps 1996), the method from algorithm theory that we extend to speed up the calculation of the heuristic values for HSP 2.0. We follow the presentation in (Ramalingam & Reps 1996). A function $g(x_1, \ldots, x_j, \ldots, x_k)$: $\mathbb{R}_+^k \rightarrow \mathbb{R}_+$ is called a strict weakly superior function (in short: swsf) if, for every $j \in 1 \ldots k$, it is monotone non-decreasing in variable x_j and satisfies: $g(x_1, \ldots, x_j, \ldots, x_k) \leq x_j \Rightarrow g(x_1, \ldots, x_j, \ldots, x_k) = g(x_1, \ldots, \infty, \ldots, x_k)$. The swsf fixed point (in short: swsf-fp) problem is to compute the unique fixed point of k equations, namely the equations $x_i = g_i(x_1, \ldots, x_k)$, in the k variables $x_1, \ldots, x_k$, where the g_i are swsf for $i = 1 \ldots k$. The dynamic swsf-fp problem is to maintain the unique fixed point of the swsf equations after some or all of the functions g_i have been replaced by other swsf's. DynamicSWSF-FP solves the dynamic swsf-fp problem efficiently by recalculating only the values of variables that change, rather than the values of all variables. The authors of DynamicSWSF-FP have proved its correctness, completeness, and other properties and applied it to grammar problems and shortest path problems (Ramalingam & Reps 1996).

Terminology and Variables

We use the following terminology to explain how to use DynamicSWSF-FP to calculate the heuristic values for HSP 2.0. If $x_i = g_i(x_1, \ldots, x_k)$ then x_i is called consistent. Otherwise it is called inconsistent. If x_i is inconsistent then either $x_i < g_i(x_1, \ldots, x_k)$, in which case we call x_i underconsistent, or $x_i > g_i(x_1, \ldots, x_k)$, in which case we call x_i overconsistent. We use the variables rhs_i to keep track of the current values of $g_i(x_1, \ldots, x_k)$. It always holds that $rhs_i = g_i(x_1, \ldots, x_k)$ (Invariant 1). We can therefore compare x_i and rhs_i to check whether x_i is overconsistent or underconsistent. We maintain a priority queue that always contains exactly the inconsistent x_i (to be precise: it stores i rather than x_i) with priorities $\min(x_i, rhs_i)$ (Invariant 2).

Transforming the Equations

We transform Equations 1 and 2 as follows to ensure that they specify a swsf-fp problem, for all $s \subseteq P$, $p \in P$ and $o \in O$:

$$g'_s(p) = \begin{cases} 0 & \text{if } p \in s \\ \min_{o \in O | p \in Add(o)}[1 + g'_s(o)] & \text{otherwise} \end{cases} \quad (3)$$

$$g'_s(o) = 1 + \sum_{p \in Prec(o)} g'_s(p). \quad (4)$$

procedure AdjustVariable(q)
if $q \in P$ then
if $q \in s$ then set $rhs_q := 0$
else set $rhs_q := 1 + \min_{o \in O | q \in Add(o)} x_o$
else /* $q \in O$ */ set $rhs_q := 1 + \sum_{p \in Prec(q)} x_p$
if q is in the priority queue then delete it
if $x_q \neq rhs_q$ then insert q into the priority queue with priority $\min(x_q, rhs_q)$

procedure SolveEquations()
while the priority queue is not empty do
delete the element with the smallest priority from the priority queue and assign it to q
if $rhs_q < x_q$ then
set $x_q := rhs_q$
if $q \in P$ then for each $o \in O$ such that $q \in Prec(o)$ do AdjustVariable(o)
else if $q \in O$ then for each $p \in Add(q)$ with $p \notin s$ do AdjustVariable(p)
else
set $x_q := \infty$
AdjustVariable(q)
if $q \in P$ then for each $o \in O$ such that $q \in Prec(o)$ do AdjustVariable(o)
else if $q \in O$ then for each $p \in Add(q)$ with $p \notin s$ do AdjustVariable(p)

procedure Main()
empty the priority queue
set s to the state whose heuristic value needs to get computed
for each $q \in P \cup O$ do set $x_q := \infty$
for each $q \in P \cup O$ do AdjustVariable(q)
forever do
SolveEquations()
/* use $h_{add}(s) = 1/2 \sum_{p \in G} x_p$ */
set $s' := s$ and s to the state whose heuristic value needs to get computed next
for each $p \in (s \setminus s') \cup (s' \setminus s)$ do AdjustVariable(p)

Figure 4: PINCH

The only difference is in the calculation of $g'_s(o)$. The transformed equations specify a swsf-fp problem since 0, $\min_{o \in O|p \in Add(o)}[1 + g'_s(o)]$, and $1 + \sum_{p \in C} g'_s(p)$ are all swsf in $g'_s(p)$ and $g'_s(o)$ for all $p \in P$ and all $o \in O$. This means that the transformed equations can be solved with DynamicSWSF-FP. They can be used to calculate $h_{add}(s)$ since it is easy to show that $g_s(p) = 1/2 g'_s(p)$ and thus $h_{add}(s) = \sum_{p \in G} g_s(p) = 1/2 \sum_{p \in G} g'_s(p)$.

Calculation of Heuristics with DynamicSWSF-FP

We apply DynamicSWSF-FP to the problem of calculating the heuristic values of HSP 2.0, which reduces to solving the swsf fixed point problem defined by Equations 3 and 4. The solution is obtained by SolveEquations() shown in Figure 4. In the algorithm, each x_p is a variable that contains the corresponding $g'_s(p)$ value, and each x_o is a variable that contains the corresponding $g'_s(o)$ value.

PINCH calls AdjustVariable() for each x_q to ensure that Invariants 1 and 2 hold before it calls SolveEquations() for the first time. It needs to call AdjustVariable() only for those x_q whose function g_q has changed before it calls SolveEquations() again. The invariants will automatically continue to hold for all other x_q. If the state whose heuristic value needs to get computed changes from s' to s, then this changes only those functions g_q that correspond to the right-hand side of Equation 3 for which $p \in (s \setminus s') \cup (s' \setminus s)$, in other words, where p is no longer part of s (and the corresponding variable thus is no longer clamped to zero) or just became part of s (and the corresponding variable thus just became clamped to zero). SolveEquations() then operates as follows. The x_q solve Equations 3 and 4 if they are all consistent. Thus, SolveEquations() adjusts the values of the inconsistent x_q. It always removes the x_q with the smallest priority from the priority queue. If x_q is overconsistent then SolveEqua-

```
procedure AdjustVariable($q$)
  if $x_q \neq rhs_q$ then
    if $q$ is not in the priority queue then insert it with priority $\min(x_q, rhs_q)$
    else change the priority of $q$ in the priority queue to $\min(x_q, rhs_q)$
  else if $q$ is in the priority queue then delete it

procedure SolveEquations()
  while the priority queue is not empty do
    assign the element with the smallest priority in the priority queue to $q$
    if $q \in P$ then
      if $rhs_q < x_q$ then
        delete $q$ from the priority queue
        set $x_{old} := x_q$
        set $x_q := rhs_q$
        for each $o \in O$ such that $q \in Prec(o)$ do
          if $rhs_o = \infty$ then
            set $rhs_o := 1 + \sum_{p \in Prec(o)} x_p$
          else set $rhs_o := rhs_o - x_{old} + x_q$
          AdjustVariable($o$)
      else
        set $x_q := \infty$
        if $q \notin s$ then
          set $rhs_q = 1 + \min_{o \in O | q \in Add(o)} x_o$
          AdjustVariable($q$)
        for each $o \in O$ such that $q \in Prec(o)$ do
          set $rhs_o := \infty$
          AdjustVariable($o$)
    else /* $q \in O$ */
      if $rhs_q < x_q$ then
        delete $q$ from the priority queue
        set $x_q := rhs_q$
        for each $p \in Add(q)$ with $p \notin s$
          $rhs_p = \min(rhs_p, 1 + x_q)$
          AdjustVariable($p$)
      else
        set $x_{old} := x_q$
        set $x_q := \infty$
        set $rhs_q := 1 + \sum_{p \in Prec(q)} x_p$
        AdjustVariable($q$)
        for each $p \in Add(q)$ with $p \notin s$
          if $rhs_p = 1 + x_{old}$ then
            set $rhs_p := 1 + \min_{o \in O | p \in Add(o)} x_o$
            AdjustVariable($p$)

procedure Main()
  empty the priority queue
  set $s$ to the state whose heuristic value needs to get computed
  for each $q \in P \cup O$ do set $rhs_q := x_q := \infty$
  for each $o \in O$ with $Prec(o) = \emptyset$ do set $rhs_o := x_o := 1$
  for each $p \in s$ do
    $rhs_p := 0$
    AdjustVariable($p$)
  forever do
    SolveEquations()
    /* use $h_{add}(s) = 1/2 \sum_{p \in G} x_p$ */
    set $s' := s$ and $s$ to the state whose heuristic value needs to get computed next
    for each $p \in s \setminus s'$ do
      $rhs_p := 0$
      AdjustVariable($p$)
    for each $p \in s' \setminus s$ do
      $rhs_p := 1 + \min_{o \in O | p \in Add(o)} x_o$
      AdjustVariable($p$)
```

Figure 5: Optimized PINCH

tions() sets it to the value of rhs_q. This makes x_q consistent. Otherwise x_q is underconsistent and SolveEquations() sets it to infinity. This makes x_q either consistent or overconsistent. In the latter case, it remains in the priority queue. Whether x_q was underconsistent or overconsistent, its value got changed. SolveEquations() then calls AdjustVariable() to maintain the Invariants 1 and 2. Once the priority queue is empty, SolveEquations() terminates since all x_q are consistent and thus solve Equations 3 and 4. One can prove that it changes the value of each x_q at most twice, namely at most once when it is underconsistent and at most once when it is overconsistent, and thus terminates in finite time (Ramalingam & Reps 1996).

Algorithmic Optimizations

PINCH can be optimized further. Its main inefficiency is that it often iterates over a large number of propositions or operators. Consider, for example, the case where $q \in O$ has the smallest priority during an iteration of the while-loop in SolveEquations() and $rhs_q < x_q$. At some point in time, SolveEquations() then executes the following loop

for each $p \in Add(q)$ with $p \notin s$ do AdjustVariable(p)

The for-loop iterates over all propositions that satisfy its condition. For each of them, the call AdjustVariable(p) executes

$$\text{set } rhs_p := 1 + \min_{o \in O | p \in Add(o)} x_o$$

The calculation of rhs_p therefore iterates over all operators that contain p in their add list. However, this iteration can be avoided. Since $rhs_q < x_q$ according to our assumption, SolveEquations() sets the value of x_q to rhs_q and thus decreases it. All other values remain the same. Thus, rhs_p cannot increase and one can recalculate it faster as follows

$$\text{set } rhs_p := \min(rhs_p, 1 + x_q)$$

Figure 5 shows PINCH after this and other optimizations. In the experimental section, we use this method rather than the unoptimized one to reduce the planning time. This reduces the planning time, for example, by 20 percent in the Logistics domain and up to 90 percent in the Freecell domain.

Summary of Methods

Table 1 summarizes the methods that we have discussed and classifies them according to whether they order the value updates (ordered updates) and whether they reuse information from the calculation of previous heuristic values (incremental computations).

Both VI and IVI perform full sweeps over all variables and thus perform unordered updates. IVI reuses the values of variables from the computation of the previous heuristic value and is an incremental version of VI. We included HSP2 although it is very similar to VI and belongs to the same class because it is the method used by HSP 2.0. Its only difference from VI is that it eliminates all operator variables, which simplifies the code.

GBF, HSP1, and GD order the value updates. They are listed from left to right in order of increasing number of binary ordering constraints between value updates. GBF performs full sweeps over all operator variables interleaved with partial sweeps over those proposition variables whose values might have changed because they depend on operator variables whose values have just changed. HSP1, the method used by HSP 1.0, alternates partial sweeps over operator variables and partial sweeps over proposition variables. Finally, both GD and PINCH order the value updates completely and thus do not perform any sweeps. This enables GD to update the value of each variable only once and PINCH to update the value of each variable only twice. PINCH reuses the values of variables from the computation of the previous heuristic value and is an incremental version of GD.

Problem Size	#P	#O	Length	#CV	#CV/#P	HSP2	VI	IVI	GBF	HSP1	GD	PINCH
LOGISTICS-4-0	48	78	26	15.10	11.98%	0.07	0.09 (-29%)	0.09 (-29%)	0.06 (14%)	0.06 (14%)	0.09 (-29%)	0.05 (29%)
LOGISTICS-7-0	99	174	52	29.43	10.78%	0.82	1.13 (-38%)	0.97 (-18%)	0.58 (29%)	0.60 (27%)	1.13 (-38%)	0.49 (40%)
LOGISTICS-10-0	168	308	59	39.73	8.35%	1.06	1.54 (-45%)	1.32 (-25%)	0.69 (35%)	0.76 (28%)	1.51 (-42%)	0.52 (51%)
LOGISTICS-13-0	275	650	102	52.48	5.67%	5.25	8.08 (-54%)	6.54 (-23%)	3.52 (33%)	3.77 (28%)	13.75 (-162%)	2.05 (61%)
LOGISTICS-16-0	384	936	121	63.33	4.80%	12.01	18.27 (-52%)	15.03 (-25%)	7.86 (35%)	8.13 (32%)	19.69 (-64%)	4.11 (66%)
LOGISTICS-19-0	511	1274	144	84.26	4.72%	30.93	43.12 (-39%)	34.36 (-11%)	18.25 (41%)	18.55 (40%)	45.45 (-47%)	9.16 (70%)
LOGISTICS-22-0	656	1664	160	108.65	4.68%	101.72	165.72 (-63%)	137.86 (-36%)	69.67 (32%)	71.64 (30%)	178.57 (-76%)	34.98 (66%)
LOGISTICS-25-0	855	2664	206	104.28	2.96%	104.12	168.14 (-61%)	128.58 (-23%)	73.15 (30%)	87.08 (16%)	188.62 (-81%)	27.99 (73%)
LOGISTICS-28-0	1040	3290	243	124.14	2.86%	201.38	316.96 (-57%)	249.55 (-24%)	140.38 (30%)	151.34 (25%)	362.90 (-80%)	51.09 (75%)
LOGISTICS-31-0	1243	3982	269	148.05	2.83%	315.95	491.25 (-55%)	382.49 (-21%)	201.35 (36%)	220.12 (30%)	546.57 (-73%)	72.86 (77%)
LOGISTICS-34-0	1464	4740	291	161.16	2.60%	434.81	688.98 (-58%)	518.06 (-19%)	296.51 (32%)	307.26 (29%)	801.60 (-84%)	105.07 (76%)
LOGISTICS-37-0	1755	6734	316	165.82	1.95%	1043.80	1663.30 (-59%)	1333.00 (-28%)	759.29 (27%)	824.53 (21%)	2132.00 (-104%)	219.82 (79%)
LOGISTICS-40-0	2016	7812	337	186.08	1.89%	1314.80	2112.70 (-61%)	1464.30 (-11%)	900.32 (32%)	936.42 (29%)	2488.30 (-89%)	275.39 (79%)

Table 2: Experimental Results in the Logistics Domain

Experimental Evaluation

After describing our experimental setup and the collected data, we discuss the results and draw conclusions.

Experimental Setup

To compare the various methods for calculating the heuristic values, we integrated them into the publicly available code for HSP 2.0.[2] We used the default configuration of HSP 2.0 for all experiments, namely forward weighted A* searches with a weight of 2.0.

HSP2 is already part of the publicly available code for HSP 2.0. We therefore used it as the baseline method against which we compared the other methods. To make this baseline as efficient as possible, we deleted all code from the existing implementation of HSP2 whose results are not used when planning with the default configuration, which reduces the planning time of HSP2, for example, by over 20 percent in the Logistics domain.

Most of our test problems came from the publicly available AIPS-98 and AIPS-00 competition problem sets. We used all instances of the Gripper, Mprime and Mystery domains from AIPS-98. In addition, we generated random problems in the Gripper domain that are larger than those from AIPS-98. We used all domains from AIPS-00: all instances of the Blocks World, Elevator, Freecell, and Logistics domains and small instances (with two to nine parts) from the Schedule domain. Finally, we generated additional problems in the Blocks World domain that are larger than those from AIPS-00. These problems, randomly drawn from a uniform distribution of problems, allow us to reliably identify a trend for large problems (these results appear in a separate graph).

We performed the experiments on a cluster of 32 Sun Sparc Ultra 10 workstations with 256 MBytes of memory each. We limited the planning time to 10 minutes for each problem.

Reported Data

Table 2 contains detailed data for the Logistics domain. These results are not directly comparable to the results from the AIPS-00 competition since HSP 2.0 solved most Logistics problems in the competition with backward search (Bonet & Geffner 2001a). The rows corresponds to problems of increasing size. The first column contains the problem name. The next two columns contain the number #P of propositions that are contained in at least one add or delete list of the applicable ground operators and the number #O of applicable ground operators, respectively. The sum of #O and #P is the size of the graph used to compute the heuristic values. The fourth column contains the length of the plan found by HSP 2.0. The fifth column contains the average number #CV of proposition variables whose values changed from the calculation of one heuristic value to the next. The sixth column contains the ratio of #CV and #P. The seventh column contains the planning time for HSP2, our baseline. Finally, the next six columns contain both the planning times of the other methods and their relative speedup over HSP2. None of the planning times include the time required for generating the propositions and ground operators because it is the same for each method.

Figure 6 contains less detailed data for all planning domains. It plots, for each domain, the relative speedup in planning time of all methods over HSP2 as a function of the size of the graph used to compute the heuristic values. (Thus, the horizontal line at y=0 corresponds to HSP2.) Each data point corresponds to a single problem. Lines average over all problems of the same size. When a problem could not be solved within 10 minutes, we approximated the relative speedup in planning time with the relative speedup in node generation rate and indicated this with a dashed line. This approximation is justified because we checked empirically that the node generation rate remains roughly constant over time.

Results and Discussion

The relative speedups of the methods in planning time over HSP2 vary substantially, both across different methods in the same domain and across different domains for the same method. However, we can draw two conclusions.

First, PINCH and GBF are the best and second best methods for large domains, respectively. Indeed, PINCH is significantly faster than GBF in five domains, about as fast as GBF in two other domains, and only significantly slower than GBF in the Freecell domain. (GBF, in turn, is significantly faster than all methods other than PINCH in all but three domains.) Furthermore, the relative speedup in plan-

[2] We followed the publicly available code for HSP 1.0 when reimplementing HSP1. All priority queues were implemented as binary heaps. The threshold on the number of sweeps for the IVI method was set to 10.

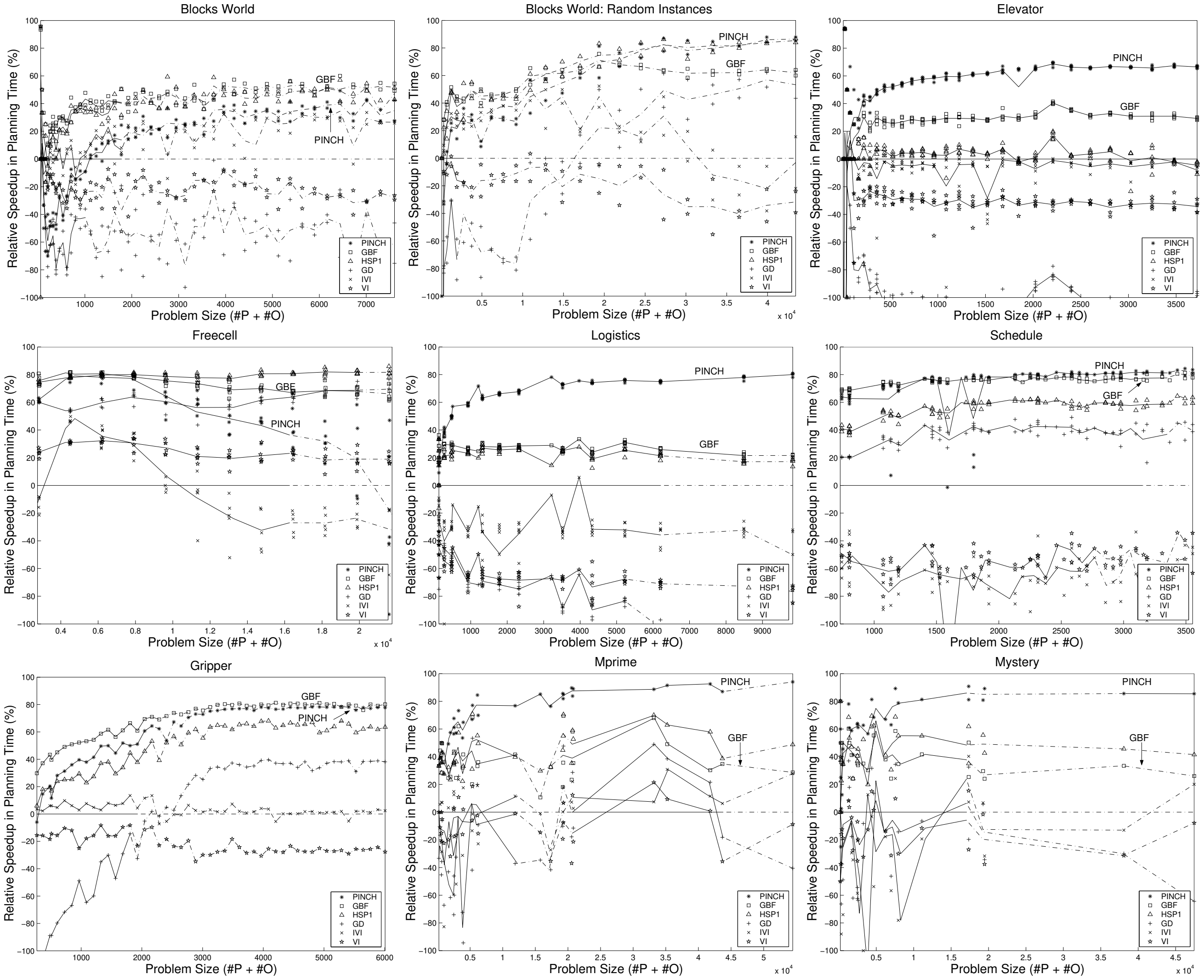

Figure 6: Relative Speedup in Planning Time over HSP2

ning time of PINCH over HSP2 tends to increase with the problem size. For example, PINCH outperforms HSP2 by over 80 percent for large problems in several domains. According to theoretical results given in (Ramalingam & Reps 1996), the complexity of PINCH depends on the number of operator and proposition variables whose values change from the calculation of one heuristic value to the next. Since some methods do not use operator variables, we only counted the number of proposition variables whose values changed. Table 2 lists the resulting dissimilarity measure for the Logistics domain. In this domain, #CV/#P is a good predictor of the relative speedup in planning time of PINCH over HSP2, with a negative correlation coefficient of -0.96. More generally, the dissimilarity measure is a good predictor of the relative speedup in planning time of PINCH over HSP2 in all domains, with negative correlation coefficients ranging from -0.96 to -0.36. This insight can be used to explain the relatively poor scaling behavior of PINCH in the Freecell domain. In all other domains, the dissimilarity measure is negatively correlated with the problem size, with negative correlation coefficients ranging from -0.92 to -0.57. In the Freecell domain, however, the dissimilarity measure is positively correlated with the problem size, with a positive correlation coefficient of 0.68. The Freecell domain represents a solitaire game. As the problem size increases, the number of cards increases while the numbers of columns and free cells remain constant. This results in additional constraints on the values of both proposition and operator variables and thus in a larger number of value changes from one calculation of the heuristic value to the next. This increases the dissimilarity measure and thus decreases the relative speedup of PINCH over HSP2. Finally, there are problems that HSP 2.0 with PINCH could solve in the 10-minute time limit that neither the standard HSP 2.0 distribution, nor HSP 2.0 with GBF, could solve. The Logistics domain is such a domain.

Second, GBF is at least as fast as HSP1 and HSP1 is at least as fast as GD. Thus, the stricter the ordering of the variable updates among the three methods with ordered updates and nonincremental computations, the smaller their relative speedup over HSP2. This can be explained with the increasing overhead that results from the need to maintain the ordering constraints with increasingly complex data structures. GBF does not use any additional data structures, HSP1 uses an unordered list, and GD uses a priority queue. (We implemented the priority queue in turn as a binary heap, Fibonacci heap, and a multi-level bucket to ensure that this conclusion is independent of its implementation.) Since PINCH without value reuse reduces to GD, our experiments demonstrate that it is value reuse that allows PINCH to counteract the overhead associated with the priority queue, and makes the planning time of PINCH so competitive.

Conclusions

In this paper, we systematically evaluated methods for calculating the heuristic values for HSP 2.0 with the h_{add} heuristics and demonstrated that the resulting planning times differ substantially. We identified two different approaches for speeding up the calculation of the heuristic values, namely to order the value updates and to reuse information from the calculation of previous heuristic values. We then showed how these two approaches can be combined, resulting in our PINCH (Prioritized, INCremental Heuristics calculation) method. PINCH outperforms both of the other approaches individually as well as the methods used by HSP 1.0 and HSP 2.0 for most of the large planning problems tested. In fact, it speeds up the planning time of HSP 2.0 by up to eighty percent in several domains and, in general, the amount of savings grows with the size of the domains. This is an important property since, if a method has a high relative speedup for small problems, it wins by fractions of a second which is insignificant in practice. However, if a method has a high relative speedup for large problems, it wins by minutes. Thus, PINCH allows HSP 2.0 to solve larger planning problems than was possible before in the same amount of time and without changing its operation. We also have preliminary results that show that PINCH speeds up HSP 2.0 with the h_{max} heuristics (Bonet & Geffner 2001b) by over 20 percent and HSP 2.0 with the h^2_{max} (Haslum & Geffner 2000) heuristic by over 80 percent for small blocks world instances. We are currently working on demonstrating similar savings for other heuristic search-based planners. For example, PINCH also applies in principle to the first stage of FF (Hoffmann & Nebel 2001a), where FF builds the planning graph.

Acknowledgments

We thank Blai Bonet and Hector Geffner for making the code of HSP 1.0 and HSP 2.0 available to us and for answering our questions. We also thank Sylvie Thiébaux and John Slaney for making their blocksworld planning task generator available to us. The Intelligent Decision-Making Group is partly supported by NSF awards to Sven Koenig under contracts IIS-9984827, IIS-0098807, and ITR/AP-0113881 as well as an IBM faculty partnership award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

References

Bertsekas, D. 2001. *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proceedings of the 5th European Conference on Planning*, 360–372.

Bonet, B., and Geffner, H. 2001a. Heuristic search planner 2.0. *Artificial Intelligence Magazine* 22(3):77–80.

Bonet, B., and Geffner, H. 2001b. Planning as heuristic search. *Artificial Intelligence – Special Issue on Heuristic Search* 129(1):5–33.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism. In *Proceedings of the National Conference on Artificial Intelligence*, 714–719.

Cormen, T.; Leiserson, C.; and Rivest, R. 1990. *Introduction to Algorithms*. MIT Press.

Dijkstra, E. 1959. A note on two problems in connection with graphs. *Numerical Mathematics* 1:269–271.

Edelkamp, S. 2001. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning*, 13–24.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, 70–82.

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

Hoffmann, J., and Nebel, B. 2001b. RIFO revisited: Detecting relaxed irrelevance. In *Proceedings of the 6th European Conference on Planning*, 325–336.

Knuth, D. 1977. A generalization of Dijkstra's algorithm. *Information Processing Letters* 6(1):1–5.

McDermott, D. 1996. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, 142–149.

Moore, A., and Atkeson, C. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning* 13(1):103–130.

Nguyen, X.; Kambhampati, S.; and Nigenda, R. S. 2002. Planning graph as the basis for deriving heuristics for plan synthesis by state space and csp search. *Artificial Intelligence* 135(1–2):73–123.

Pearl, J. 1985. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Ramalingam, G., and Reps, T. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21:267–305.

Refanidis, I., and Vlahavas, I. 2001. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research* 15:115–161.