A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm

Jörg Hoffmann
Institute for Computer Science
Albert Ludwigs University
Am Flughafen 17
79110 Freiburg, Germany
hoffmann@informatik.uni-freiburg.de

January 14, 2000

Technical Report No. 133

Abstract

The standard approach to obtain a heuristic is to relax the problem $\mathcal{P}$ at hand into a simpler problem $\mathcal{P}'$. The optimal solution length for situations in $\mathcal{P}'$ can then be used as an admissible estimate for the difficulty of situations in $\mathcal{P}$. Applying this to domain independent planning, one encounters serious problems: a planning problem $\mathcal{P}$ can easily be relaxed into $\mathcal{P}'$ by ignoring the delete lists of the operators, but, as the authors of HSP point out, computing the length of an optimal solution plan for $\mathcal{P}'$ is still NP-hard.

What one *can*, however, compute efficiently, is *some* plan that solves $\mathcal{P}'$. In this paper, we introduce a method for doing so. The relaxed plans obtained this way give a good estimate for the length of a real solution, and they can also be used in a simple manner to guide action selection during planning. Using these informations, excellent results can be obtained by employing a simple search strategy that proceeds towards the goal in a greedy manner. The Algorithm we propose is complete on deadlock-free domains. With a greedy search strategy, it can not guarantee the solution plans to be optimal. In spite of this, it finds optimal, or close to optimal, plans in most cases. Often, it solves the problems almost without any search at all.

1 Introduction

The standard approach to obtain a heuristic is to relax the problem $\mathcal{P}$ at hand into some easier problem $\mathcal{P}'$. The optimal solution length to a situation in $\mathcal{P}'$ can then be used as an admissible estimate for the optimal solution length of the same situation in $\mathcal{P}$. An application of this idea to domain independent planning was first used in the HSP system, as was first described in [Bonet et al., 1997]. The planning problem $\mathcal{P}$ is relaxed by simply ignoring the delete lists of all operators. However, computing the optimal solution length for a planning problem without delete lists is still NP-hard, as was first shown in [Bylander, 1992]. Therefore, the HSP heuristic is only a rough estimate of the relaxed solution length. In short, it is obtained by summing up the minimal distances of all atomic goals.

In this paper, we go one step further. We introduce a method that computes *some*, not necessarily optimal, solution to the relaxed problem. These solutions are helpfull in two ways:

- in most cases, their length provides an informative estimate for the difficulty of a situation;
- often, they are pretty close to a real solution, so one can use them as a guidance for action selection.

In some planning domains, like the *gripper* problems from the 1998 AIPS planning systems competition or the *tyreworld*, it is sufficient to use this heuristic in a naive *Hill-Climbing*-like strategy. Starting from the initial state of one of these problems, one can pick, in each state, any successor state with minimal heuristic value, and ends up with an optimal solution plan. This surprising informativeness of the heuristic on some domains leads us to an algorithm that proceeds towards the goal in a greedy manner. Unlike most planning algorithms, it does not search until it finds a valid plan from the initial to the goal state, but selects certain plan prefixes and never takes those decisions back.

Again surprisingly, this method shows stable behaviour over almost all domains that are commonly used in the planning community. In particular, we will see that it is complete on the class of *deadlock-free* domains, and that almost all commonly used domains belong to that class. Employing a greedy search strategy, the method can, of course, not guarantee its solution plans to be optimal. In spite of this, it finds optimal, or close to optimal, plans in most cases. As a benefit from the severe restriction of its search space, it shows very competitive runtime behaviour. For example, the *gripper* and *logistic* problems from the planning competition are solved faster than by any other domain independent planning system known to the author at the time of writing.

The rest of the paper is organized as follows. In the next section, we briefly review the basic definitions of STRIPS problems. Section 3 is concerned with the algorithms that extract a relaxed solution to a certain situation. In Section 4, we explain in detail the greedy search strategy that makes use of the heuristic in an effective manner, and Section 5 gives the proof of the strategys completeness on deadlock-free problems. In Section 6, a way is shown in which relaxed solutions can be used to guide action selection during planning, and Section 7 exemplifies the runtime behaviour of the method on a variety of well known planning problems. Related literature is discussed in Section 8, where we also compare our heuristic to the HSP heuristic, and the paper concludes with the outlook in Section 9.

2 Background

Throughout the paper, we consider simple STRIPS domains, as were first introduced in [Fikes and Nilsson, 1971]. In the following, we briefly review the basic definitions.

Definition 1 (State) *The set of all ground atoms is denoted with P. A* state *$s \in 2^P$ is a set of ground atoms.*

We assume that all operator schemata are grounded, i.e., we only talk about *actions*.

Definition 2 (Strips Action) *A* STRIPS action *o has the usual form*

$$o = [\, pre(o) \;\Rightarrow\; ADD \;\; add(o) \;\; DEL \;\; del(o) \,]$$

where $pre(o)$ are the preconditions of o, $add(o)$ is the Add list of o and $del(o)$ is the Delete list of the action, each being a set of ground atoms. The result of applying a STRIPS action to a state is defined as usual:

$$Result(s, o) = \begin{cases} (s \cup add(o)) \setminus del(o) & pre(o) \subseteq s \\ s & otherwise \end{cases}$$

In the first case, where $pre(o) \subseteq s$, the action is said to be applicable *in s. The result of applying a sequence of more than one action to a state is recursively defined as*

$$Result(s, \langle o_1, \ldots, o_n \rangle) = Result(Result(s, \langle o_1, \ldots, o_{n-1} \rangle), o_n).$$

Definition 3 (Planning Problem) *A* planning problem *$\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$ is a triple where $\mathcal{O}$ is the set of actions, and $\mathcal{I}$ (the initial state) and $\mathcal{G}$ (the goals) are sets of ground atoms.*

Our heuristic method is based on *relaxed planning problems*, which are defined as follows.

Definition 4 (Relaxed Planning Problem) *Given a planning problem $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$. The* relaxed *problem $\mathcal{P}'$ to $\mathcal{P}$ is defined as $\mathcal{P}' = (\mathcal{O}', \mathcal{I}, \mathcal{G})$, with*

$$\mathcal{O}' = \{[\ pre(o)\ \Rightarrow\ ADD\ \ add(o)\ \ DEL\ \ \emptyset\ ]\ |\ [\ pre(o)\ \Rightarrow\ ADD\ \ add(o)\ \ DEL\ \ del(o)\ ] \in \mathcal{O}\}$$

In words, one obtains the relaxed planning problem by ignoring the delete lists of the operators. We define plans simply to be *sequences* of actions (rather than sequences of parallel steps), as the algorithm we propose does not search for parallel plans, unlike, for example, the GRAPHPLAN system [Blum and Furst, 1997].

Definition 5 (Plan) *Given a planning problem $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$. A* plan *$P$ is a sequence $P =< o_1, \dots, o_n >$ of actions in $\mathcal{O}$, $o_i \in \mathcal{O}$ for $1 \leq i \leq n$. A plan P is a* solution *to $\mathcal{P}$, iff $\mathcal{G} \subseteq Result(\mathcal{I}, P)$. A plan is called a* relaxed solution *to $\mathcal{P}$, iff it solves the relaxed planning problem $\mathcal{P}'$ to $\mathcal{P}$.*

3 Heuristic

In this section, we introduce a method for heuristically evaluating planning states S. Basically, the method consists of two parts.

1. First, the *relaxed fixpoint* is built on S. This is a forward chaining process that determines in how many steps, at best, a fact can be reached from S, and with which actions.

2. Then, a *relaxed solution* is extracted from the fixpoint. This is a sequence of actions that achieves the goal from S, if their delete effects are ignored.

The first part corresponds directly to the heuristic method that is used in HSP, see [Bonet et al., 1997] or [Bonet and Geffner, 1999]. The second part goes one step further: while in HSP, the heuristic is extracted as a side effect of the fixpoint, we invest some extra effort to find a relaxed plan, and use the plan to determine our heuristic value. We will discuss related literature in section 8. For now, we have a closer look at the fixpoint process, as depicted in Figure 1.

The fixpoint process can be seen as building a layered graph structure, where fact and action layers are interleaved in an alternating fashion. The first layer, F_0, is a fact layer which is set to $F_0 := S$. These are the facts that can be made TRUE in 0 steps. Then, the first action layer O_0 gets inserted. It consists out of all actions that are applicable in S;

$F_0 := S$
$max := 0$
while $\mathcal{G} \not\subseteq \bigcup_{i=0,\dots,max} F_i$ **do**
    $O_{max} := \{o \in \mathcal{O} \mid pre(o) \subseteq \bigcup_{i=0,\dots,max} F_i\} \setminus \bigcup_{i=0,\dots,max-1} O_i$
    $F_{max+1} := \bigcup_{o \in O_{max}} add(o) \setminus \bigcup_{i=0,\dots,max} F_i$
    if $F_{max+1} = \emptyset$ **then**
        $max := \infty$
        break
    endif
    $max := max + 1$
endwhile

Figure 1: The algorithm that computes the relaxed fixpoint on a planning state S.

these are the actions whose preconditions can be achieved in 0 steps. We now obtain the second fact layer F_1 by taking all the add effects of actions in O_0, and substracting those facts that are already present in F_0; this way, we obtain axctly those facts which can be made TRUE in at least one step. Then, in the next iteration, the second action layer is inserted, leading to the third fact layer and so forth. The process terminates if

1. all goals are reached, i.e., all goal facts appear in some fact layer, or
2. the new fact layer is empty, i.e., we did not reach any new facts, which implies that the next action layer will be empty, too; the fixpoint is reached.

If the second case applies, then the fixpoint is reached before all goal facts are present. This implies unsolvability of the relaxed problem.

Proposition 1 *Given a planning problem $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ and a state S. If, after termination of the algorithm in Figure 1, $max = \infty$, then the relaxed problem to $(\mathcal{O}, S, \mathcal{G})$ is unsolvable.*

Proof Sketch: Say there is a relaxed solution P' to $(\mathcal{O}, S, \mathcal{G})$. Then, per induction over the length of P', all actions in P' appear in some action level. Each goal g is either already TRUE in S, or it is added by at least one action $o \in O_i$ out of P'. In the first case, $g \in F_0$ holds, in the second case, g is either already a member of one of the layers $F_0, \dots, F_i$, or it appears in F_{i+1}. Thus, all the goals appear in some fact layer, which is a contradiction. ■

Upon termination of the fixpoint algorithm, max tells us the layer number where the last goal came in. If the fixpoint is reached without all goals being present, this number is, consequently, set to ∞. The crucial information that the fixpoint process gives us, and that we will use to extract a relaxed plan, are the *levels* of all facts and actions; these are defined as the number of the fact- or action layer they are members of.

$$\text{level}(f) := \begin{cases} i & \text{ex. } i : f \in F_i \\ \infty & \text{otherwise} \end{cases} \tag{1}$$

$$\text{level}(o) := \begin{cases} i & \text{ex. } i : o \in O_i \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

We now show how to extract a plan from the fixpoint structure. This is done in a backward chaining manner, where we simply use any action with minimal level to make a goal TRUE. The exact algorithm is depicted in Figure 2. Note that we *do not need to search*, we can proceed right away to the initial state and are guaranteed to find a solution.

for $i := 1, \ldots, max$ **do**
    $G_i := \{g \in \mathcal{G} \mid \text{level}(g) = i\}$
endfor
$h := 0$
for $i := max, \ldots, 1$ **do**
    for all $g \in G_i$, g not marked TRUE at time i **do**
        select an action o with $g \in add(o)$ such that $\text{level}(o) = i - 1$
        $h := h + 1$
        for all $f \in pre(o)$, $\text{level}(f) \neq 0$, f not marked TRUE at time $i - 1$ **do**
            $G_{\text{level}(f)} := G_{\text{level}(f)} \cup \{f\}$
        endfor
        for all $f \in add(o)$ **do**
            mark f as TRUE at times $i - 1$ and i
        endfor
    endfor
endfor

Figure 2: The algorithm that extracts a relaxed solution to a state S after the fixpoint has been built.

The plan extraction algorithm makes use of an array of goal sets $G_1, \ldots, G_{max}$ (Of course, the algorithm is only invoked if $max < \infty$, cf. Proposition 1). All goals are inserted into the goal set that corresponds to their fact layer. Goals at level 0 are not considered, as these are members of S and do not need to be made TRUE. The mechanism then proceeds down from layer max to layer 1, and selects an action o for each goal g at the current layer i, incrementing the plan length counter h (no actions are selected for goals that are marked TRUE at the time being, as is explained later). Note that o is required to have level$(o) = i - 1$. This is minimal as the goal g has level i, i.e., the first action that achieved g in the fixpoint came in at level $i - 1$. The preconditions of o are inserted as new goals into their corresponding goal sets. If the current layer is i, then the levels of o's preconditions are at most $i - 1$, so these new goals will be made TRUE later during the process. As for the notion of "selecting" an action o: currently, an arbitrary action with the needed properties gets selected. One might imagine that selection heuristics could be applied here. We come back to this in Section 9.

One subtelty arises from the parallelity of actions: the plan extraction algorithm selects, at each layer, a *set* of actions to make the goals TRUE. The actions at one layer can, in principle, be used in any order to make up a sequential solution plan.[1] However, some orderings may be better than others, i.e., lead to shorter solutions. Let us see a simple example for that.

state S**:** $\emptyset$
goals: C, D
actions: **op1**: $\emptyset \Rightarrow ADD\ A$
op2: $\emptyset \Rightarrow ADD\ B$
op3: $A \Rightarrow ADD\ C,\ B$
op4: $B \Rightarrow ADD\ D$

Facing this scenario, the fixpoint computation will end up with $F_0 = \emptyset$, $O_0 = \{op1, op2\}$, $F_1 = \{A, B\}$, $O_1 = \{op3, op4\}$ and $F_2 = \{C, D\}$. Both goals are in the goal set at max level 2. Hence, at level 1 the actions $op3$ and $op4$ are selected. This leads to the new goals A and B at level 1, for which, in turn, the actions $op1$ and $op2$ are selected. We end up with the parallel plan $< \{op1, op2\}, \{op3, op4\} >$. Turning this into a sequential plan, it makes a difference how we order $op3$ and $op4$. One alternative is $< op1, op2, op3, op4 >$, the other one is $< op1, op2, op4, op3 >$. Having a closer look on the first alternative, one realizes that the action $op2$ has become superfluent: the precondition B of $op4$, its only reason for being in the plan, gets made TRUE by $op3$ already. Thus, if we order $op3$ before $op4$, we can achieve a shorter plan than the other way round.

[1] Readers who are used to work with GRAPHPLAN (like the author) might get familiar feelings at this point. In fact, there is a strong connection between the heuristic method shown here and GRAPHPLAN, as we will see in section 8.

The plan extraction mechanism as shown in Figure 2 orders the actions in a parallel set in the order they are inserted. This is done by marking the add effects of an action o, which achieves goals at level i, as TRUE at time $i-1$. Then, if another parallel action o' needs an add effect of o as precondition, then o—which is ordered before o' in the final sequential plan—is considered as its achiever and the goal is not made TRUE explicitly, unless an action in a lower level needs it as precondition. In the above example, if $op3$ gets selected before $op4$, then B never becomes a goal at level 1, and $op2$ is never selected. Finally, add effects of a selected action can, naturally, be also marked TRUE at time i, as these facts are already achieved and do not need a second achiever, in the case they are goals.

We now simply chain the two parts, fixpoint computation and plan extraction, together and arrive at the algorithm which evaluates a planning state S. The algorithm can be seen in Figure 3.

build the relaxed fixpoint on S, see Figure 1
if $max = \infty$ **then**
  return ∞
endif
extract a relaxed solution, see Figure 2
return h

Figure 3: The overall algorithm to evaluate a given planning state S.

There are a few things to be remarked about the plan extraction mechanism. First of all, the actions it selects do in fact make up a relaxed solution for the problem $(\mathcal{O}, S, \mathcal{G})$. We do not prove this here, as it is a bit tedious and not really important for the rest of the paper. After all, we are only using the extracted plans as a *heuristic*. This would not apply if we wanted to develop an *admissible* heuristic. In principle, this could be done by finding the *optimal* relaxed solution. This, however, is NP-hard [Bylander, 1992]. Nevertheless, it makes sense to extract *one* relaxed solution from the fixpoint. The h values obtained this way are informative in most cases—as is exemplified in section 8, and indicated by the empirical results in Section 7—, and the extracted plan itself can also be used for further guidance of the planning process, as we explain in section 6. At this point, we remark that the heuristic has the desirable property of returning 0 on, and only on, goal states.

Proposition 2 *Given a planning problem $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ and a state S. The algorithm shown in Figure 3 evaluates S to $h = 0$ if and only if $\mathcal{G} \subseteq S$.*

Proof Sketch: The algorithm returns $h = 0$ iff it does not need to select any action. This is the case, if and only if all goals are already contained in F_0, as any goal in a higher fact layer would cause at least one action to be selected. All goals being in F_0, in turn, is the same as all goals being in S, i.e., $\mathcal{G} \subseteq S$. ■

4 Search

So far, we developed a method for the heuristic evaluation of planning states. Now, we consider how to use it in a forward chaining state space planner. Following the argumentation in this section, one might wonder why nothing is said about standard heuristic search strategies like A^* or its relatives. We did not investigate in that direction so far(leave it as an open research avenue, see section 9), the reason being the following. It turns out that, using the heuristic shown in section 3, one can obtain excellent results with surprisingly simple search strategies. Most striking, one can actually find plans, in some cases even optimal plans, with a search method as simple as the one shown in Figure 4.

$S := \mathcal{I}$
obtain h by evaluating S, see Figure 3
while $h \neq 0$ **do**
    evaluate all successor states $S' = Result(S, < o >)$, for $o \in \mathcal{O}$, $pre(o) \subseteq S$
    pick a state $S' = Result(S, < o >)$ with minimal evaluation h'
    $S := S'$, $h := h'$
endwhile

Figure 4: An extremely simple greedy search method that shows surprisingly good behavior in some domains.

The strategy shown in Figure 4 is something like the most naive use of a heuristic that one can possibly imagine. Corresponding to a naive *Hill-Climbing* strategy without any restart techniques whatsoever, it simply evaluates all successor states of the current state, and picks one with minimal evaluation. It relies completely on its heuristic being good enough, following it blindly in the hope of finding a goal state. Of course, such a method is neither complete nor does it return optimal solutions. Nevertheless, it shows surprisingly good behaviour in some domains, like, for example, the *gripper* problems from the 1998 AIPS planning systems competition, and the well known *tyreworld*. In these domains, the heuristic as presented in section 3 is seemingly good enough to follow it blindly, and naive Hill-climbing does, in fact, find the optimal solutions. Like one can imagine from the pseudocode in Figure 4, this takes almost no computation time, as the algorithm does not really perform any search. [2]

Of course, naive Hill-climbing does *not* find optimal solutions to all planning problems. In most cases, it gets stuck in an infinite loop and does not return any plan at all. Neither, given a certain problem $\mathcal{P}$, is it possible to tell in advance, if the method will find a solution or not, even if $\mathcal{P}$ is known to be solvable. On the other hand, the method proves to be extremely efficient for some problems, and one would like to carry over as much of that efficiency as possible to a more stable planning algorithm. The key question that leads us to such an algorithm is:

When is it adequate to simply pick some successor state ?

And the obvious first guess answer to this question might be:

If the evaluation of that successor state is better than the evaluation of the current state.

Working on this basic idea, it is straightforward to arrive at the algorithm shown in Figure 5.

initialize the curent plan to the empty plan <>
$S := \mathcal{I}$
obtain h by evaluating S, see Figure 3
if $h = \infty$ **then**
  output "No Solution", stop
endif
while $h \neq 0$ **do**
  search for a state S' with evaluation $h' < h$
  if no such state can be found **then**
    output "No Solution", stop
  endif
  add the actions on the path to S' at the end of the current plan
  $S := S'$, $h := h'$
endwhile

Figure 5: The Enforced Hill-climbing algorithm.

[2] For example, naive Hill-climbing solves the 42-ball *gripper* problem to optimality in 0.13 seconds; in the AIPS planning systems competition, only HSP was able to solve that problem, using 33 seconds to find a suboptimal solution.

At each stage of the search, Enforced Hill-climbing picks some successor of the current state. Unlike in standard *Hill-Climbing*, this successor does not need to be a direct one, and, unlike in standard *Hill-Climbing*, we do not pick any best valued successor, but *enforce* the successor to be one that is *better* than our current state.

More precisely, the algorithm works as follows. The current situation S is initialized to the initial state. It is then evaluated heuristically, as described in section 3. If its heuristic value is ∞, then the problem is unsolvable: if the relaxed problem is not solvable (Proposition 1), then neither is the original problem. Otherwise, the algorithm invokes a search process for some state S' with better evaluation than S. Here, any complete search method would do the job; we chose to use breadth first search in our implementation, as this search strategy finds the shallowest solution, and as search spaces are usually small enough to keep them in memory. The exact search algorithm is depicted in Figure 6.

initialize empty search space
for all $o \in \mathcal{O}$, $pre(o) \subseteq S$ **do**
    add $Result(S, o)$ at end of search space
endfor
while search space not empty **do**
    S' := remove front of search space
    evaluate S' to h', see Figure 3
    if $h' < h$ **then**
        return S', h'
    endif
    for all $o \in \mathcal{O}$, $pre(o) \subseteq S'$ **do**
        add $Result(S', o)$ at end of search space
    endfor
endwhile
return $\perp$, ∞

Figure 6: A standard breadth first search for a better state S'.

This is nothing but a standard breadth first search, stopping when it either hits a state that has a better heuristic evaluation than S, or when the search space becomes empty. In the first case, the appropriate state and heuristic are returned, Enforced Hill-climbing adds the actions on the path from S to S' at the end of the current plan, and the process continues with the new start state S'. In the second case, no state is reachable from S that has a better evaluation. In particular, this means that no *goal* state is reachable from S, as the evaluation of any goal state is 0 (Proposition 2). Such, the problem we are facing right now (reaching a goal state from S) is unsolvable. There are two possible reasons for that:

1. The planning problem $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$ at hand is unsolvable, so the search must at some point hit a state S which can not be improved.

2. We did something wrong in reaching S, i.e., $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ is solvable, but $(\mathcal{O}, \mathit{Result}(\mathcal{I}, P), \mathcal{G})$ is not, where P is the current plan with $S = \mathit{Result}(\mathcal{I}, P)$.

We will in the next section identify the class of *deadlock-free* planning problems, where always the first reason applies. This gives us completeness of Enforced Hill-climbing on this class of problems.

5 Completeness

In this section, we

1. formally define what a *deadlock* in a planning problem is, and give a simple proof that Enforced Hill-climbing is complete on problems that do not contain such deadlocks;

2. argue that almost all benchmark planning domains, which attempt to model "realistic" problems, are in fact deadlock-free.

The whole argumentation follows closely, and is to some extent identical to, the definitions and results that are given in [Koehler and Hoffmann, 2000].

Definition 6 (Deadlock) *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning problem. A reachable state S is called a* deadlock *iff there is no sequence of actions that leads from S to the goal, i.e., iff $S = \mathit{Result}(\mathcal{I}, P)$ and $\neg\exists\, P' :\ \mathcal{G} \subseteq \mathit{Result}(S, P')$.*

Naturally, a planning problem is called *deadlock-free*, if it does not contain any deadlock, i.e., if there is no sequence of actions that leads from the initial state to a state where the goal can not be reached anymore. We remark that a deadlock-free problem is also solvable, cause otherwise the initial state itself would already be a deadlock.

Theorem 1 *Let $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning problem. If $\mathcal{P}$ is deadlock-free, then the* Enforced Hill-climbing *algorithm, as defined in Figure 5, will find a solution.*

Proof: There are two cases in which the Enforced Hill-climbing algorithm does not find a solution:

1. the initial state is evaluated to $h = \infty$;

2. for some intermediate state S, no better evaluated state S' can be found.

In the first case, we know with Proposition 1 that the relaxed problem $\mathcal{P}'$ to $\mathcal{P}$ is unsolvable. Of course, this implies unsolvability of $\mathcal{P}$, which is a contradiction.

In the second case, the algorithm considers some intermediate state $S = Result(\mathcal{I}, P)$, where P is the current plan, and does not find a state with better evaluation than S. Now, we know three things:

1. The evaluation h of S is $h > 0$ (otherwise, the algorithm would already have stopped).

2. The evaluation of any goal state is 0 (as shown in Proposition 2).

3. Breadth-first search is complete.

Together, we can conclude that there is no sequence of actions that leads from S to a goal state, in other words, there does not exist a plan P' with $\mathcal{G} \subseteq Result(S, P')$. Thus, S is a deadlock state in contradiction to the assumptions. ■

It should be remarked, that, with Theorem 1, it is still possible that Enforced Hill-climbing runs into an infinite loop on an intermediate state S, i.e., that it does not terminate on an unsolvable problem. In fact, following precisely the definitions that are given in section 4, this can happen. However, the only thing one needs to do is to keep a memory for past states during breadth-first search, and avoid repeated states. As state spaces in STRIPS planning are finite, this modification will make Enforced Hill-climbing stop on unsolvable problems. This seems to be of little practical value, though; searching the whole state space is, of course, unfeasible for larger scale planning problems.

Following [Koehler and Hoffmann, 2000], we will now argue that almost all commonly used planning domains are deadlock-free and can therefore be solved by Enforced Hill-climbing without the risk of loosing a solution. First, we define the class of *invertible* planning problems.

Definition 7 (Invertible Planning Problem) *Let $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ be a planning problem, and let S denote the states that are reachable from $\mathcal{I}$ with actions from $\mathcal{O}$. The problem is called* invertible *if and only if*

$$\forall\, S: \;\; \forall\, P: \; \exists\, \overline{P}: \; Result(Result(s, P), \overline{P}) = s$$

The class of solvable invertible planning problems is a subclass of the class of deadlock-free planning problems.

Proposition 3 *Given an invertible planning problem $\mathcal{P} = (\mathcal{O}, \mathcal{I}, \mathcal{G})$. If $\mathcal{P}$ is also solvable, then it does not contain any deadlocks.*

Proof Sketch: Say $S = Result(\mathcal{I}, P_S)$ is a deadlock state. Let P be an arbitrary solution to $\mathcal{P}$, and let $\overline{P_S}$ be the action sequence that inverts P_S, according to Definition 7. The concatenation of $\overline{P_S}$ with P solves $(\mathcal{O}, S, \mathcal{G})$, which is a contradiction. $\blacksquare$

Proposition 3, and the following results, are taken out of [Koehler and Hoffmann, 2000], so we give only short proof sketches here. The propositions are almost obvious; anyway, for full formal proofs, we refer the interested reader to the article named above.

So far, we know that being solvable and invertible is a sufficient condition for a planning problem to be deadlock-free. Now, we determine a sufficient condition for a planning problem to be invertible. It is then easily seen that almost all commonly used benchmark domains fullfill this condition. We are talking about the existence of *inverse actions.*

Definition 8 (Inverse Action) *Given an action set $\mathcal{O}$ containing an action o of the form $pre(o) \Rightarrow add(o)\ del(o)$. An action $\overline{o} \in \mathcal{O}$ is called* inverse *to o if and only if $\overline{o}$ has the form $pre(\overline{o}) \Rightarrow add(\overline{o})\ del(\overline{o})$ and satisfies the following conditions*

1. $pre(\overline{o}) \subseteq pre(o) \cup add(o) \setminus del(o)$
2. $add(\overline{o}) = del(o)$
3. $del(\overline{o}) = add(o)$

Lemma 1 *Given a state S and an action o with $del(o) \subseteq pre(o)$ that is applicable in S. An action $\overline{o}$ that is inverse to o in the sense of Definition 8 is applicable in $Result(S, <o>)$ and $Result(Result(S, <o>), <\overline{o}>) = S$ holds.*

Proof Sketch: The Lemma is easily proven by inserting the appropriate Definitions 2 and 8 of the *Result* function and inverse actions, respectively. The condition $del(o) \subseteq pre(o)$ is needed as otherwise, $\overline{o}$ might "re-add" facts to the state that have not been there in the first place. $\blacksquare$

If we have an inverse action to each action o in a planning problem, then the problem is invertible.

Proposition 4 *Given a planning problem $(\mathcal{O}, \mathcal{I}, \mathcal{G})$ with the set of ground actions $\mathcal{O}$, where for each action $del(o) \subseteq pre(o)$ holds. If there is an inverse action $\overline{o} \in \mathcal{O}$ for each action $o \in \mathcal{O}$, then the problem is invertible.*

Proof Sketch: Given a state S in which an action sequence $P =< o_1, \ldots, o_n >$ is carried out, one gets back to S by executing $\overline{P} :=< \overline{o_n}, \ldots, \overline{o_1} >$. This can be proven by inductively applying Lemma 1. ∎

Putting Proposition 4, Proposition 3 and Theorem 1 together, we know that Enforced Hill-climbing is complete on planning problems where

1. an inverse action exists to each action in $\mathcal{O}$, and
2. each action fullfills $del(o) \subseteq pre(o)$ (needed by Lemma 4).

Both conditions are fullfilled in a natural way by the most planning problems.

As for the second condition, one finds this phenomenon in *all* domains that are commonly used by the planning community, at least in those that are known to the author. It is just something that seems to hold in any reasonable logical problem formulation. Some authors even need it as an assumption for their algorithms to work, like [Fox and Long, 1998].

As for the existence of inverse actions, one usually finds them in all planning domains which attempt to model realistic (although simplified) problems. For example in the *blocksworld*, we have **stack** and **unstack**. Similarly in all domains that deal with logistics problems, for example *logistics*, *mystery*, *rocket*, *gripper* etc., one can always find inverse pairs of actions. Sometimes, two different ground instances of the same operator schema yield an inverse pair. For example, in *gripper*, the two ground instances

move(roomA, roomB)
at-robby(roomA) ⇒ ADD at-robby(roomB) DEL at-robby(roomA).

and

move(roomB, roomA)
at-robby(roomB) ⇒ ADD at-robby(roomA) DEL at-robby(roomB).

of the **move(?from,?to)** operator schema invert each other.

Only very rarely, non-invertible actions occur in standard domains. If they do, their role in the domain is often quite limited. A nice example are the operators **cuss** and **inflate** in Russel's Tyreworld.

cuss
⇒ *DEL annoyed().*

inflate(?x:wheel)
have(pump) not-inflated(?x) intact(?x) ⇒ *ADD inflated(?x) DEL not-inflated(?x).*

Obviously, there is not much point in defining something like a **decuss** or a **deflate** operator. More formally speaking, none of the ground actions to these operators destroys a goal or a precondition of any other action in the domain. Therefore, it does not matter that their effects cannot be inverted. In particular, these actions do not lead into deadlocks. [3]

6 Helpful Actions

Up to this point, we have used the relaxed solutions, as are computed by the machinery presented in Section 3, only in terms of their length, to estimate the cost of a state S. They are, however, often pretty close to a real solution; in particular, one can use them to determine a set of actions that seem to be *helpful* for reaching the goal from a state S. Let us exemplify this with a small example taken from the *gripper* domain.

We do not repeat the exact definition of the *gripper* domain here, as it is easily understood intuitively: There are two rooms, A and B, and a certain number of balls, which are all in room A initially and shall be moved into room B. The planner controls a robot, which changes rooms via the **move** operator, and which has two grippers to **pick** or **drop** balls. Each gripper can hold only one ball at a time. We look at a small problem where 2 balls must be moved into room B.

A relaxed solution to the initial state that our heuristic might extract is

< { **pick** ball1 A left,
pick ball2 A left,
move A B },
{ **drop** ball1 B left,
drop ball2 B left } >

This is a parallel relaxed plan consisting out of two time steps. Note that the **move** A B action is selected parallel to the **pick** actions, as the relaxed planner does not notice that it can not **pick** balls in room A anymore once it has moved into room B. In a similar

[3] The **cuss** operator, by the way, is the only one known to the author that deletes a fact it is not using as a precondition. It is also the only one I know that could be removed from the domain description without changing anything.

fashion, both balls are picked with the left gripper: both actions have a *free(left)* fact as precondition, which they delete, but the relaxed planner does not care about the deletes. Note also that, in spite of these simplifications, the length of the relaxed plan, 5, is exactly the length of an optimal solution plan.

We will now focus on the actions that are selected in the *first time step* of the relaxed plan, with the intention to restrict the number of actions that the planner has to try in the current state S. For the time being, we restrict the planner to try *only* those actions that are contained in the first step of the relaxed plan. In the example, two of these are first steps in an optimal solution, namely both **pick** actions. Say the planner considers the action **pick** ball1 A left. In the resulting state, the relaxed solution looks like this.

< { **pick** ball2 A right,
move A B },
{ **drop** ball1 B left,
drop ball2 B right } >

Note here that, again, the length of this plan is exactly the optimal solution length.[4] As the left gripper already holds ball1, ball2 gets correctly picked with the free right gripper. This time, the first plan step contains two actions, of which the **pick** action leads to an optimal solution. Considering this action, we get to a state where the first relaxed plan step contains only *one* action.

< { **move** A B },
{ **drop** ball1 B left,
drop ball2 B right } >

Now, following the first relaxed plan steps leads us directly to a solution: after selecting **move** A B, the relaxed solution consists only out of the two **drop**s, and after selecting one of these, only the other one is left to select, and we are finished.

To avoid confusion: the above example is *not* aimed at illustrating any hypothetical planning process; the point we want to make is this. Restricting the action choice in a state S to the first relaxed plan step, any planning process on the *gripper* example does not consider useless actions like **drop**ping balls in room A or **pick**ing balls in room B. In general, the actions in the first relaxed plan step are often those which lead in the direction of the goal, or, in other words, which are helpful in the current state. However, considering only those actions that are part of the relaxed solution is a bit too restrictive in some cases. Consider the following *blocksworld* example.

[4] In fact, when used in the correct order, the actions in this relaxed plan already make up a solution to our problem. We take up this thought in Section 9.

Say we use the well known *blocksworld* representation with four operators, **stack**, **unstack**, **pickup** and **putdown**. The planner controls a single robot arm, and the operators can be used to **stack** one block on top of another one, **unstack** a block from another one, **pickup** a block from the table, or **put** a block that the arm is holding **down** onto the table. Initially, the arm is holding block C, and blocks A and B are on the table. The goal is to stack A onto B. There are three different shortest relaxed solutions.[5]

< { **putdown** C },
  { **pickup** A },
  { **stack** A B } >

or

< { **stack** C A},
  { **pickup** A },
  { **stack** A B } >

or

< { **stack** C B},
  { **pickup** A },
  { **stack** A B } >

All of these are valid relaxed solutions, as in the relaxation it does not matter that **stack**ing C onto A or B deletes facts that we still need. If C is on A, we can not **pickup** A anymore, and if C is on B, we can not **stack** A onto B anymore.

The first action in each relaxed plan is only inserted to get rid of C, i.e., free the robot arm, and from the point of view of the relaxed planner, all of the three starting actions do the job. So the relaxed solution that is extracted by our heuristic might be any of the three above. If it happens to be the second or third one, then we loose the path to an optimal solution by restricting ourselfs to the corresponding actions, **stack** C A or **stack** C B. Therefore, we define the set H of helpful actions that we consider as follows.

$$H := \{o \in O_0 \mid add(o) \cap G_1 \neq \emptyset\}$$

Remember, after extraction of a relaxed solution, O_0 contains all actions that where applicable in S, and G_1 contains all the facts that where the goals to be achieved at time

[5]These are the *shortest* solutions in terms of the number of time steps. The relaxed plan that our heuristic extracts is always one of the plans that are shortest in this sense.

step 1. In words, we consider as helpful actions all those, which have at least one goal at the first time step as an add-effect. In the above *blocksworld* example, freeing the robot arm is among these goals, which causes all the three starting actions to be elements of H. In the *gripper* example we considered before, the sets H defined this way are exactly the actions in the first time step of the relaxed plan, so the modification does not change anything there.

To make clear how usefull the set H of helpful actions can be, consider one last example. Say we have a *blocksworld* problem, where hundreds of blocks are on the table initially, but the goal is only to stack one block A on top of another block B. A naive algorithm doing forward search will in this case suffer from the high branching factor; even if the algorithm uses a heuristic that detects the right successor state, the heuristic evaluation of all successors must be costly. Using the set H, which in this case contains only the one action **pickup** A, a planner can step on to the best state right away. In some sense, the helpful actions mechanism detects the large amount of *irrelevance* in the problem. We will discus the connection to irrelevance detection mechanisms like [Nebel et al., 1997] in Section 8. Now, we show how to integrate the helpful actions heuristic into our planning algorithm.

build the relaxed fixpoint on S, see Figure 1
if $max = \infty$ **then**
  return ∞
endif
extract a relaxed solution, see Figure 2
$H := \{o \in O_0 \mid add(o) \cap G_1 \neq \emptyset\}$
return h, H

Figure 7: The algorithm that evaluates a given planning state S, and collects a set H of promising actions to try in the next step.

First of all, we update the planning state evaluation mechanism, Figure 3, to not only return the heuristic value of a state, but also the set H of actions that are considered helpful. See the definition in Figure 7.

Throughout our experiments, the set H almost always contained at least one action which lead from S to an optimal solution. Thus, it seems there is not much danger of loosing a solution if one restricts the planner to *only* consider the helpful actions for the successor states. In fact, we found only very few (solvable) examples where one can not find a solution that way. In the case that this happens, we simply remove the restriction and let the planner search the complete search space. See the updated Enforced Hill-climbing algorithm in Figure 8, and the heuristic search algorithm in Figure 9.

initialize the curent plan to the empty plan <>
$S := \mathcal{I}$
obtain h and H by evaluating S, see Figure 7
if $h = \infty$ **then**
  output "No Solution", stop
endif
while $h \neq 0$ **do**
  heuristic search for a state S' with evaluation $h' < h$, Figure 9
  if no such state can be found **then**
    complete search for a state S' with evaluation $h' < h$, Figure 4
    if no such state can be found **then**
      output "No Solution", stop
    endif
  endif
  add the actions on the path to S' at the end of the current plan
  $S := S'$, $h := h'$
endwhile

Figure 8: The Greedy Heuristic Improvement algorithm, updated to make use of the helpful actions heuristic.

Figures 8 and 9 together make up almost the same search algorithm as was described in Section 4. The only difference is that now, on each intermediate state S, the planner first tries to find a better state S' by doing *heuristic* search. Heuristic search here does *not* mean that the search algorithm picks states according to some heuristic value—as usual—, but that the search algorithm *restricts the successor states* according to some heuristic, namely the helpful actions heuristic. The pseudocode given in Figure 9 should be understood easily. Instead of adding *all* successors to the search space after evaluating a state, the breadth first search adds only those that seem to be helpful.

Given the case that this way the search space gets empty, i.e., no solution can be found, the updated Enforced Hill-climbing algorithm, Figure 8, simply switches back to complete breadth first search. It should be noted here that, to make this strategy work, at least the heuristic breadth first search *needs* a memory for past states. Otherwise, it will most likely run into an infinite loop on problems that it can not solve, and the complete search will never be invoked. Given such a memory, however, it is obvious that the revised Enforced Hill-climbing algorithm still preserves completeness on deadlock-free planning problems, as was proven in Section 5. We will see in Section 7 that it shows

```
initialize empty search space
for all o ∈ H do
    add Result(S, o) at end of search space
endfor
while search space not empty do
    S' := remove front of search space
    evaluate S' to h' and H', see Figure 7
    if h' < h then
        return S', h'
    endif
    for all o ∈ H' do
        add Result(S', o) at end of search space
    endfor
endwhile
return ⊥, ∞
```

Figure 9: Heurstic breadth first search for a better state S', considering only helpful actions.

dramatically improved runtime behaviour compared to Enforced Hill-climbing without the helpful actions heuristic.

7 Empirical Results

For empirical evaluation, we implemented the updated Enforced Hill-climbing algorithm, using relaxed plans to evaluate states and to determine helpful actions, in C. We call the resulting planning system FF, which is short for FAST-FORWARD planning system.[6] The name is meant to reflect the way in which the Enforced Hill-climbing algorithm proceeds towards the goal, running as fast in a forward direction as it can. All running times for FF are measured on a Sparc Ultra 10 running at 350 MHz, with a main memory of 256 MBytes. Where possible, the running times of other planners where measured on the same machine. Otherwise, they are from the literature where comparable machines with at least 300 MHz have been used. We indicate runtimes taken from the Literature in the text.

[6] It is also something my father always said to me when he wanted to say "Good Luck".

7.1 The *gripper* domain

One of the domains where FF does extraordinarily good is the *gripper* domain. The domain was used in the 1998 AIPS planning systems competition. It is a very simple domain, where there are two rooms, a robot with two grippers, and a number of balls that need to be transported from one room into the other room. Despite this simplicity, the domain gives rise to an exponential explosion in search space, which is the reason why no planner in the competition solved problems with more than 8 balls except for HSP. By the time being, five planners are known to the author that solve all the *gripper* problems from the competition: HSP[Bonet et al., 1997], HSP-r[Bonet and Geffner, 1999], STAN[Long and Fox, 1999], GRT[Refanidis and Vlahavas, 1999], and FF. Four out of these five are heuristic planners. The only exception, GRAPHPLAN-based STAN, makes use of a sophisticated symmetry and domain analysis machinery. Two planners, namely FF and HSP-r, solve the problems substantially faster than the others. The biggest problem is solved by HSP, STAN and GRT in 33, 10 and 5 seconds, respectively. Both of the other planners need only less than a second. That is why Figure 10 shows the runtime curves only for these planners.

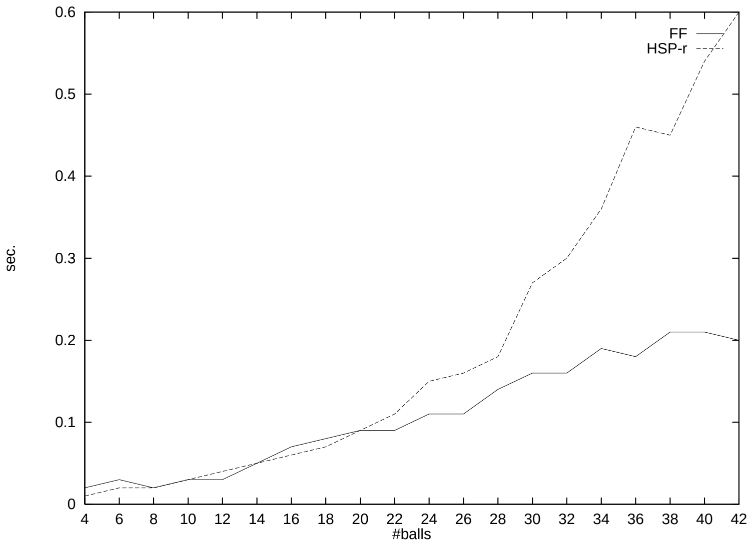

Figure 10: Runtime curves of HSP-r and FF on *gripper* problems of increasing size.

As can be seen in Figure 10, both planners solve all the problems in almost no time. While being a bit faster sometimes on smaller problems, HSP-r is three times slower than FF on the biggest problem. It seems that HSP-r does not scale to big instances quite as good as FF.

The reason for the efficiency of FF on these problems is that it needs to evaluate only a very small number of states. As a matter of fact, the number of evaluated states is a linear function of the problem size. Precisely, in our current implementation, FF looks at $n/2 * 7$ states, where n is the number of balls to be transported. In the biggest example, only 147 states are evaluated to find an optimal solution plan of length 125.

7.2 The *logistics* Domain

This is a classical domain, involving the transportation of packets via trucks and airplanes. Trucks are usually restricted to move within a city, while airplanes move between the airports of different citys. There are two well known test suites. One has been used in the planning competition, the other one is part of the BLACKBOX distribution. The problems in the competition suite are very hard. In fact, they are so hard that, up to date, no planner has been reported to solve them all. FAST-FORWARD is the first one that does. See Tabular 11, showing also the results for GRT and HSP-r, which are the only other planners—known to the author—standing a chance on these problems. In all tabulars, a dash indicates that the corresponding planner failed to find a plan on that problem within half an hour.

	HSP-r		GRT		FF	
problem	time	steps	time	steps	time	steps
prob-01	0.36	35	0.28	30	0.06	27
prob-02	3.13	36	1.32	34	0.19	32
prob-03	25.45	64	5.55	60	0.71	54
prob-04	50.13	63	19.28	69	0.98	58
prob-05	0.62	27	0.39	26	0.08	22
prob-06	293.60	83	14.39	80	1.95	73
prob-07	6.20	37	1.76	37	0.38	36
prob-08	-	-	16.37	48	2.04	41
prob-09	371.03	97	50.48	98	2.08	91
prob-10	287.64	121	23.13	117	3.20	103
prob-11	4.58	34	1.54	36	0.21	30
prob-12	-	-	43.06	48	2.01	41
prob-13	-	-	85.58	79	7.73	67
prob-14	-	-	60.20	104	6.97	98
prob-15	19.52	120	67.50	106	1.27	93
prob-16	92.75	69	31.58	62	1.23	55
prob-17	29.35	61	12.19	53	0.63	44
prob-18	-	-	335.05	193	50.76	167
prob-19	-	-	238.98	174	16.26	151
prob-20	-	-	324.12	169	24.40	139
prob-21	-	-	294.23	120	8.93	102
prob-22	-	-	-	-	246.05	282
prob-23	100.67	145	16.86	118	3.84	126
prob-24	-	-	98.54	49	4.17	40
prob-25	-	-	-	-	106.23	181
prob-26	-	-	-	-	71.15	183
prob-27	-	-	-	-	71.26	141
prob-28	-	-	-	-	679.43	265
prob-29	-	-	-	-	589.75	323
prob-30	-	-	-	-	62.4	131

Figure 11: Results of the three best *logistic* planners on the competition problem suite. Times are in seconds, *steps* counts the number of actions in a sequential plan.

The times for GRT in Tabular 11 are taken out of [Refanidis and Vlahavas, 1999], where they are measured on a Pentium 300 with 64 MByte main memory. The other times are taken on our own slightly faster machine with more main memory.

However, space consumption of FF is almost marginal (a fixed search space size of 10000 nodes was sufficient to solve all of the problems in our experiments), and anyway, it can be clearly seen from Tabular 11 that FF outperforms both HSP-r and GRT by an order of magnitude. Also, it finds shorter plans than the other planners.

We do not show runtimes here for the BLACKBOX distribution suite. This would be a similar big tabular, showing similar results. We ran FF on these problems, and it solved all of them in less than half a second. Compared to the results shown in [Bonet and Geffner, 1999] for these problems, FF was between 2 and 10 times faster than HSP-r, finding shorter plans in all cases.

We also ran FF on the *mystery* and *mprime* examples from the competition, which are encoded *logistic* domain variations. FF solves 14 problems of the *mystery* suite, and 18 problems of the *mprime* suite. For comparison: the best results reported for any planner in terms of solved problems are 15 for STAN on the *mystery*, and 15 for HSP on the *mprime* problems. On most of the examples that it solves, FF finds the solution in less than a second. On one *mystery* problem, it needs more time, namely 4.81 seconds on *prob-20*. For the hardest to solve *mprime* problem, 8.49 seconds are required.

7.3 Mixed classical Problems

FAST-FORWARD also shows competitive behaviour on a variety of other commonly used benchmark domains. We show a tabular of running times in Figure 12, comparing FF against a collection of state-of-the-art planning systems: IPP [Koehler et al., 1997], STAN [Long and Fox, 1999], BLACKBOX [Kautz and Selman, 1999], and HSP [Bonet et al., 1997].

In Tabular 12, the planning problems shown are the following. The *tyreworld* problem was originally formulated by Stuart Russell, and asks the planner to replace a flat tire. The problem is modified in a natural way so as to make the planner replace n flat tires. FF is the only planner that is capable of replacing more than three tires, scaling up to much bigger problems.

The *hanoi* problems make the planner solve the well known *Towers of Hanoi* problem, with n discs to be moved. Note here that, though running times for FF are still good on the shown problems, we definetely have a strong exponential growth in the number of discs. Now, it is no big deal to be exponential on the *Tower of Hanoi* problems, as even the solution length is exponential in the number of discs. Nevertheless, we remark an oddity that occurs in the *hanoi* domain with respect to relaxed plans. Those do not make a good goal distance estimate in these problems: for example, with 9 discs, the distance

		IPP		STAN		BLACKBOX		HSP		FF	
domain	problem	time	steps	time	steps	time	steps	time	steps	time	steps
tyreworld	fixit-1	0.04	19	0.10	19	0.43	19	1.19	19	0.04	19
tyreworld	fixit-2	11.29	30	1.25	30	114.32	30	-	-	0.09	30
tyreworld	fixit-3	-	-	-	-	933.14	41	-	-	0.20	41
tyreworld	fixit-4	-	-	-	-	-	-	-	-	0.42	52
tyreworld	fixit-5	-	-	-	-	-	-	-	-	0.95	63
hanoi	tower-3	0.03	7	0.03	7	0.23	7	0.15	7	0.01	7
hanoi	tower-5	0.11	31	0.27	31	680.6	31	162.01	31	0.09	31
hanoi	tower-7	1.93	127	6.10	127	-	-	-	-	0.52	127
hanoi	tower-9	39.31	511	230.20	511	-	-	-	-	6.45	511
bulldozer	bull-1	0.04	9	0.08	9	0.29	9	0.22	9	0.01	9
ferry	ferry-1	0.01	7	0.01	7	0.1	7	0.03	7	0.02	7
manhattan	mh-7	4.82	35	20.04	35	-	-	1260.01	40	0.12	43
manhattan	mh-11	65.12	40	1013.96	40	-	-	-	-	0.26	48
manhattan	mh-15	-	-	-	-	-	-	-	-	0.79	64
manhattan	mh-19	-	-	-	-	-	-	-	-	1.70	90

Figure 12: Running times and quality (in terms of number of actions) of plans for FF and state-of-the-art planners on various classical domains.

from the initial state to the goal is estimated to 9, while it is, in fact, 511. The *hanoi* domain is the only domain where we found such drastical differences between the real and the estimated distances of a state.

The *bulldozer* and *ferry* domains are frequently used in the Literature. Both problems used in our experiments are quite small, and neither of them causes serious trouble to any of the planners.

The *manhattan* domain was first introduced by Drew McDermott in [McDermott, 1996]. In these problems, the planner controls a robot which moves on a $n \times n$ grid world. There are three different keys which fit into three different kinds of locks, and the goal is to transport one of the keys to a certain point on the grid. To achieve this goal, a different key must be used to open the surrounding locks of the goal position. The original problem taken from [McDermott, 1996] corresponds to the *mh-11* entry in Tabular 12, where the robot moves on a 11×11 grid. The other entries refer to problems that have been modified to encode 7×7, 15×15 and 19×19 grid worlds, respectively. This gives us a series of *manhattan* problems of increasing size.[7] FF is the only planner that can handle all of them. The other planners are either not even able to solve the smallest instance, or undergo a vast exponential explosion in search space. Note that FF finds slightly suboptimal plans, but offers drastically better solution times than the other planners.

Looking at Tabular 12, one might wonder where the *blocksworld* problems are. We show them in a separate tabular, as the behaviour of FF on them needs some closer investigation. Results on the SATPLAN examples from [Kautz and Selman, 1996] are shown in Tabular 13.

[7] All problems used in these experiments are avaiable via E-Mail to the author.

	IPP		STAN		BLACKBOX		HSP		FF	
problem	time	steps	time	steps	time	steps	time	steps	time	steps
bw-large-a	0.29	12	0.40	12	1.74	12	101.4	16	0.02	12
bw-large-b	18.16	18	30.60	18	97.75	18	-	-	2.91	30
bw-large-c	-	-	-	-	-	-	-	-	-	-
bw-large-d	-	-	-	-	-	-	-	-	13.03	56

Figure 13: Running times and quality of plans on the *blocksworld*, using a representational variant with the four operators **stack**, **unstack**, **pickup** and **putdown**.

The *blocksworld* shown in Tabular 13 uses a common representational variant with the four operators **stack**, **unstack**, **pickup** and **putdown**. The behaviour of FF on this domain is not really bad, but a bit strange: while to the other problems, a plan is found quite fast, no plan at all is found in reasonable time for *bw-large-c*. This has, to the best of our knowledge, two reasons.

1. In the *blocksworld*, lots of goal orderings exist. These can be misleading for the relaxed planner, as it does not recognize them.

2. In the four operator variant, the single robot arm is coded by the 0-ary *arm-empty()* fact. This is misleading for a relaxed planner: once it has achieved this fact, it will never think about it again, and thus use the same arm to **unstack**, for example, a whole bunch of blocks without putting them away.

Let us first have a look at the second problem. It does not arise in a *blocksworld* problem, where the planner controls several arms. We modified the **stack**, **unstack**, **pickup** and **putdown** operators to use as an additional parameter a robot arm, and we provided the *bw-large-c* example with 2, 3, 4 and 5 robot arms to use. See the results for FF in Tabular 14.

arms	time	steps
2	29.30	48
3	11.40	42
4	2.07	30
5	1.06	26

Figure 14: Running times and quality of plans for FF on the *bw-large-c* example, using an increasing amount of robot arms.

Tabular 14 nicely shows that FF does better when a lot of robot arms are avaiable. In contrast to this, classical planning algorithms have been recently criticised to worsen in behaviour, when they are given more resources [Srivastava and Kambhampati, 1999]. In particular, this is the case for GRAPHPLAN.

We wanted to investigate how the other planners from our experiments react to an increasing amount of resources. For the sake of shorter experimentation time, we ran the planners on instances of the easier problem *bw-large-a*, modified to provide an increasing number of robot arms. See the results in Tabular 15.

	IPP		STAN		BLACKBOX		HSP		FF	
arms	time	steps	time	steps	time	steps	time	steps	time	steps
2	1.11	12	0.86	12	3.87	12	11.33	16	0.11	12
3	1.60	12	1.87	12	4.46	12	2.79	20	0.12	12
4	1.98	12	0.80	14	2.14	12	1.87	12	0.11	12
5	2.25	12	0.6	12	1.17	12	2.22	14	0.12	12

Figure 15: Running times and quality of plans on the *bw-large-a* example, using an increasing amount of robot arms.

Some things in Tabular 15 are as expected, other things were quite a surprise to us. We expected that, with a growing number of robot arms, IPP worsens in behaviour (as is predicted by [Srivastava and Kambhampati, 1999]), that FF is unimpressed (the problem is already easy with only one arm) and that HSP improves (as heuristic search should be easier with a lot of arms). We did not expect that STAN and BLACKBOX do better with more arms. We do not have an explanation for that, and it seems contrary to the argumentation given in [Srivastava and Kambhampati, 1999], as both of these planners are, in a way, GRAPHPLAN-descendant.

Anyway, we have seen that FF can easily solve hard *blocksworld* problems when a sufficient number of arms is avaiable. However, one might want to solve a *blocksworld*, or similar, problem with only one robot arm. It turns out that FF suits this task much better when we use a different *representational variant* of the *blocksworld*.

So far, we have used a common representational variant of the *blocksworld* with four operators. It turns out that the behaviour of FF improves dramatically when one uses the other common representational variant with only *one* operator. See Tabular 16.

	IPP		STAN		BLACKBOX		HSP		FF	
problem	time	steps	time	steps	time	steps	time	steps	time	steps
bw-large-a	0.47	10	0.57	10	10.30	10	2.55	13	0.08	10
bw-large-b	2.20	14	4.04	14	160.14	14	2.12	12	0.22	14
bw-large-c	88.17	25	267.08	26	-	-	11.71	18	2.69	22
bw-large-d	362.19	33	-	-	-	-	37.29	29	5.36	29

Figure 16: Running times and quality of plans on the *blocksworld*, using a representational variant with only one **move** operator.

In the other representational variant of the *blocksworld*, used in Tabular 16, only one **move** operator exists, which either moves a block A from one block B onto another block C, or moves some block A from another block B to the table, or moves a block A from the table onto some block B. For the STRIPS translation of this variant, one needs to define three operators. The distinction between the table and a block in the one-operator variant makes use of a conditional efffect.

It is clearly seen from Tabular 16 that FF behaves much better on this representational variant, or at least more stable. This is because in this variant, there is no 0-ary *arm-empty()* fact that can disturb the relaxed planner.

The plan lengths here seem to be shorter than with the four-operator variant. This is only because each action in the one-operator variant corresponds to a composed **unstack-stack**, **pickup-stack** or **unstack-putdown** action. As a matter of fact, the operators of the four-operator variant must always be used in one of these three combinations. We are currently working on a mechanism that automatically produces such composed operators in domains where this is possible, and thus avoids the representational *blocksworld* problem altogether.

The other problem that the relaxed planner may have with the *blocksworld* are, as pointed out above, the goal orderings. This is, in short, explained as follows. If we have to build a huge stack of blocks, this is encoded by several *on(X Y)* predicates. Now, having reached one of these predicates, a relaxed planner might think that it is closer to the goal. This is not necessarily the case: Say the planner has stacked one block on top of the other, which both shall be near the top of the final stack. If the bottom of the stack has not yet been built, this stacking action is of no use. Situations like this are often not recognised by a relaxed plan.

Now, there have been several approaches to deal with goal orderings in the Literature. We implemented the approach that is discussed in [Koehler and Hoffmann, 2000]. Here, orderings are detected heuristically as a preprocess to planning, and used to split the goal set into a series of smaller sets, which are then fed to the planner one by one. This also solves the problems that FF has with the four-operator variant of the *blocksworld*, see the running times in Tabular 17.

7.4 Benefits of Helpful Actions

We now give some empirical data to show the benefits that FF draws from using the *helpful actions* heuristic. To do this, we count the number of *evaluated states*, i.e., the number of states that are evaluated by the Enforced Hill-climbing algorithm before it finds a solution. Results on a variety of problems are listed in Tabular 18.

As Tabular 18 clearly shows, the planner can draw benefits from the *helpful actions*

problem	time	steps
bw-large-a	0.03	14
bw-large-b	0.05	22
bw-large-c	3.38	48
bw-large-d	2.03	54

Figure 17: Running times and quality of plans for FF on the *blocksworld* four-operator variant, when using a heuristic preprocessing method to cope with goal orderings.

heuristic on a whole variety of domains. On all but one of the problems from our test suite, it needs to evaluate less states when using the heuristic. This strongly indicates that the actions that are considered as helpful are, in most cases, in fact those that put the planner closer to the goal.

Another observation indicating this is the following. Restricting itself to consider *only* the helpful actions, the planner could, in principle, easily miss good ways of getting to the goals. More specifically, if the optimal action in a situation (the action leading to an optimal solution) was not considered helpful, then the planner could not find it, using the heuristic. However, in all but one cases, the plans found when using the heuristic are not longer—sometimes even shorter—than those that are found when planning without the heuristic. Thus, it seems that the helpful actions heuristic rarely ever misses an action that *is* helpful in a given situation.[8]

Looking at Tabular 18, one observes differences between the domains wrt. the *amount* of effort that the planner can save when using helpful actions. The heuristic seems to be good on *bulldozer*, *ferry*, *blocksworld* and *hanoi* problems. It works excellent on the *gripper* and *tyreworld* domains, and it yields drastical savings on *manhattan* and *logistics*. One can explain this intuitively as follows. The heuristic is extracted from a relaxed solution, so the closer that relaxed solution is to a real solution, the better would one expect the heuristic to do. In fact, relaxed plans are not much of a help on *hanoi* problems, while they are very often almost identical to real solutions in *logistics* problems.

One can directly address the effectiveness of the helpful actions heuristic by measuring

[8] It might seem odd that there are cases when Enforced Hill-climbing with helpful actions finds shorter plans than without. This can happen due to the greedy search strategy. Quite a number of a states successors can have an improved heuristic value, and the algorithm will pick the first one it finds. Other successors might be better for reaching the goals, though. Thus, the behaviour of the algorithm depends on the—more or less—random order in which it evaluates the successors of a state. Using helpful actions, the algorithm might randomly pick a successor that is better than the one that is found when evaluating *all* successors of a state.

domain	problem	FF \ H states	FF \ H steps	FF states	FF steps
gripper	4 balls	32	11	14	11
gripper	42 balls	1115	125	147	125
logistics	competition prob-01	871	27	45	27
logistics	competition prob-02	3360	33	88	32
tyreworld	fixit-1	173	19	67	19
tyreworld	fixit-2	788	30	160	30
tyreworld	fixit-3	2058	41	324	41
hanoi	tower-3	20	7	15	7
hanoi	tower-5	155	31	122	31
hanoi	tower-7	1406	127	1088	127
bulldozer	bull-1	51	10	38	9
ferry	ferry-1	17	7	8	7
manhattan	mh-7	777	42	101	42
manhattan	mh-11	1890	48	99	48
manhattan	mh-15	4735	76	157	64
blocksworld-one-op	bw-large-a	73	10	38	10
blocksworld-one-op	bw-large-b	191	16	88	14
blocksworld-one-op	bw-large-c	491	21	522	22
blocksworld-one-op	bw-large-d	1307	31	672	29

Figure 18: FF without the *helpful actions* heuristic (on the left) compared to FF using that heuristic (on the right). *states* counts the number of planning states that the heuristic is applied to before a solution is found, *steps* counts the number of actions in the found solution plan. The *blocksworld* problems use the one-operator representational variant, the *logistics* problems are taken from the planning systems competition suite.

the *percentage* of applicable actions that are selected as helpful in a given situation. We did this, and averaged the percentage over all situations that are evaluated when solving a planning problem. The results are quite in line with the overall results shown in Tabular 18. The best percentage values are obtained in *logistics* problems, where only around 10% of the applicable actions are considered heplfull on average. Thus, the branching factor for FF is *ten times smaller* on *logistics* problems when using helpful actions. In contrast, the percentage converges to 100% on *hanoi* problems, i.e., it amounts to 80% with three discs to move, and goes up to 99% on the seven discs problem. On the other problems from Tabular 18, around 50% of the actions are considered helpful, which still halves the branching factor for breadth first search. On the manhattan domain, the percentage decreases for problems of increasing size, going down from 60% on the 7×7 grid world to 44% on the 19×19 grid.

8 Related Work

The closest relative to the work described in this paper is, quite obviously, the HSP system, as was first described in [Bonet et al., 1997]. HSP was the first heuristic search planning system that attracted considerable attention in the planning community, probably because of its success in the 1998 AIPS planning systems competition. In short, HSP does heuristic forward search, with the heuristic function defined as follows.

$$h(S) := \sum_{g \in \mathcal{G}} weight(g)$$

where the *weight* of a fact f is

$$weight(f) := \begin{cases} 0 & f \in S \\ m & min\{\sum_{p \in pre(o)} weight(p) \mid o \in \mathcal{O}, f \in add(o)\} = m - 1 \\ \infty & otherwise \end{cases}$$

The weights of all facts are obtained by doing exactly the same fixpoint computation as is described in Section 3, updating the weight of a fact each time an action becomes applicable that adds it. The main problem in HSP is that the heuristic needs to be recomputed for each single search state, which is very time consuming. Inspired by HSP, a few approaches have been developed that try to cope with this problem, like HSP-r[Bonet and Geffner, 1999] and the GRT-planner [Refanidis and Vlahavas, 1999].

The authors of HSP themselves handle the problem by sticking to their heuristic, but changing the search direction, going backwards from the goal in HSP-r instead of forward from the initial state in HSP. This way, they need to compute a weight value for each fact only once, estimating the facts distance to the initial state, and only sum the weight values up for a state later during search.

The authors of [Refanidis and Vlahavas, 1999] invert, in some way, the direction of the HSP *heuristic* instead. While HSP computes distances going from the current state towards the goal, GRT goes from the goal to each fact, and estimates its distance. The function that then extracts, for each state during forward search, the states heuristic estimate, uses the precomputed distances as well as some information on which facts will probably be achieved simultaneously. This gives very good performance on a variety of domains.

For the FAST-FORWARD planning system, which we describe in this paper, a somewhat paradox extension of HSP has been made. Instead of avoiding the major drawback of the HSP strategy, we even worsen it, at first sight: the heuristic keeps being fully recomputed for each search state, and we even put some extra effort on top of it, by extracting a relaxed solution from the fixpoint. Surprisingly, the resulting planning system is, for a lot of planning domains, the most efficient solver known to the author at the time of writing. In particular, it is, in most cases, a good deal more efficient than the HSP, GRT and HSP-r approaches to heuristic search planning. We have presented empirical evidence for this in Section 7.

The reasons for the good behaviour of the FF system are—as the author suspects—the following.

1. The extra effort for extracting relaxed solutions from the fixpoint is marginal.
2. By using the helpful actions heuristic, the planner usually gets straight to the best successor states.

As for the first point, we do not show a tabular or something else indicating this here. It is quite intuitve, as computing the fixpoint is only bounded by the number of actions and facts in the domain, while extracting the relaxed solution is bounded by the number of *goals* (and subgoals) that are achieved in the plan. As for the second point, we presented empirical results indicating this in Section 7.

We remark here that there are some domains where the length of a relaxed plan is a much better distance estimate than the HSP heuristic. On the other hand, there are also domains where the HSP heuristic seems to be more appropriate, or where both heuristics return more or less the same values. In Tabular 19, we show, for a variety of planning problems, the real distance from the initial state to the goals, as well as the HSP estimate of this distance, and the FF estimate of this distance.

domain	problem	real	HSP	FF
gripper	4 balls	11	12	9
gripper	42 balls	125	126	85
logistics	competition prob-01	27	31	24
logistics	competition prob-02	32	29	28
tyreworld	fixit-1	18	46	12
tyreworld	fixit-2	30	92	20
tyreworld	fixit-3	41	138	28
hanoi	tower-3	7	3	3
hanoi	tower-5	31	5	5
hanoi	tower-7	511	7	7
bulldozer	bull-1	9	9	5
ferry	ferry-1	7	6	5
manhattan	mh-7	35	28	19
manhattan	mh-11	40	32	21
blocksworld-one-op	bw-large-a	6	12	10
blocksworld-one-op	bw-large-b	9	21	15
blocksworld-one-op	bw-large-c	18	49	20
blocksworld-one-op	bw-large-d	25	63	25

Figure 19: Real solution length, HSP estimate and FF estimate on a variety of planning problems.

Tabular 19 shows that neither the FF heuristic nor the HSP heuristic is clearly superior to the other. While the FF estimate is far better in the *tyreworld* and *blocksworld* domains, HSP gives better estimates on *gripper* and *manhattan*. On *hanoi*, both heuristics are exactly the same, on the *logistics* problems, they are more or less the same. Considering this, it is unclear why FF outperforms HSP on all those problems (except for the easy *bulldozer* and *ferry* instances). One reason surely is the *helpful actions* heuristic employed by FF, as we have already seen in Section 7.4. As an interesting piece of work, it remains to investigate the effect of the search strategies on the behaviour of the HSP and FF algorithms. One should try to use the HSP heuristic in the Enforced Hill-climbing algorithm, and one should try to use the FF heuristic in the HSP search algorithm. One could also try to use a combination of both heuristics in either search algorithm. We did not yet have the time to do any of this, as it requires a lot of implementation work.

Turning our look on the way how a relaxed solution is computed to a situation S, the similarity to what GRAPHPLAN[Blum and Furst, 1997] does becomes apparent. First, starting from S, fact and action layers are alternated in a forward chaining manner until all goals ar reached for the first time. Then, in a backward chaining manner, a solution is extracted using that layer information. Looking closer at this similarity, we found, to our surprise, that the relaxed plans one gets this way are *exactly the same* that GRAPHPLAN would compute, when started on the relaxed problem $\mathcal{P}'$ to $P = (\mathcal{O}, S, \mathcal{G})$.[9] This is relatively easy to see, at least for someone who is used to work with GRAPHPLAN. As the problem is relaxed, no exclusion relations whatsoever can be inferred, and the planning graph structure corresponds directly to our fact and action layers. Then, during backward search, standard GRAPHPLAN makes use of the "noops-first" strategy, which means something like: "Achieve each goal as far down in the graph as possible". Now, this is exactly what we do during relaxed plan extraction.

Thus, the heuristic evaluation of a state, as described in Section 3, can be seen as an application of GRAPHPLAN to a special class of planning problems, namely those without delete lists. This might seem a bit odd, as, of course, noone would try to start GRAPHPLAN on each search state during a forward search and use its output for heuristic evaluation. In fact, the extraction of relaxed solutions is a *very* special case from the point of view of GRAPHPLAN, and can be implemented far more efficiently.

There is another, less striking, parallelity between the FF and the GRAPHPLAN system. Traditionally, planning was attacked by searching backwards from the goals, in order to focus the search on actions that are relevant for solving the problem. However, employing a forward chaining phase *before* searching backwards, GRAPHPLAN achieves much better runtime behaviour than traditional backward planners. This is mostly due to the so-called exclusion relations that GRAPHPLAN infers, which can prune many unreachable goal sets

[9]More precisely, the found plans depend on which action is selected to achieve each goal or subgoal. The *set* of plans that *might be found* is the same for both methods. This is not true for all cases, as our method might detect a linearization of parallel steps that leads to a shorter solution, cf. Section 3.

during backward search.

The heuristic that we introduce in this paper, in parallel to GRAPHPLAN, uses a forward chaining method before it starts "searching" backwards for a relaxed solution. One could also try, in parallel to traditional planners, to search for a relaxed plan backwards from the goal right away. In fact, there have been approaches like this in the literature. In [McDermott, 1996], Greedy Regression-Match Graphs are introduced. In a nutshell, these estimate the goal distance of a state by backchaining from the goals, ignoring any conflicts, and then counting the estimated minimal number of steps that are needed to achieve the goal state. The backward chaining process can be exponential, in principle, but is reported to behave polynomial in practice, i.e., in the experiments carried out for [McDermott, 1996].

To the best of our understanding, the action chains that lead to a states heuristic estimate in [McDermott, 1996] are somewhat similar to the relaxed plans that we extract. However, the backchaining process is, though usually not exponential, quite costly. For example, the *manhattan* world 11×11 grid problem, that we have already seen in Section 7.3, is the main test example in [McDermott, 1996]. To build the Greedy Regression-Match Graph for the initial state of this problem, the implementation is reported to take 25 seconds on a Sparc 2 station. For comparison, we ran FF on a Sparc 4 station. Finding a relaxed plan for the initial state of the 11×11 *manhattan* problem takes less than one hundredth of a second on that machine, i.e., the time measured is 0.00 CPU seconds.

Thus, it seems that, when trying to find a relaxed plan by backward search, it is a good idea to employ a forward chaining phase *before* backward search starts, in parallel to the efficiency gains that were reported for GRAPHPLAN. In relaxed planning, the efficiency gains are not due to the pruning of unreachable states, but rather to the information about the *distance* of a fact or action to the initial state. As we have already seen in Section 3, always picking an action with minimal distance to achieve a fact leads to a relaxed plan without any real search at all.

We conclude our discussion of related work with a few thoughts on the *helpful actions* heuristic. The method shares some similarities with what is known as *relevance* from the literature, for example [Nebel et al., 1997]. The main difference is that relevance in the usual sense refers to what is usefull for solving the whole problem. Being helpful, on the other hand, refers to something that is usefull *in the next step.* This has the disadvantage that the helpful things need to be recomputed for each search state, but the advantage that possibly far less things are helpful than are relevant. In our method, we are computing the relaxed plans for each state anyway, which gives us the helpful actions for free, so we are not facing the disadvantage in our specific setting.

Talking about methodolgy, the only approach known to the author where it is tried to find actions that are usefull in a given situation are the Greedy Regression-Match Graphs discussed above. During the regression process, actions are collected that are *favored* by the regression graph. This, in short, means that they are applicable in the given state and that they are part of the action chain that leads to the states heuristic estimate. The method is quite different from the way that we obtain our set of helpful actions, and it takes much longer computation time, as argued above.

9 Outlook

In this paper, we presented two heuristics for domain independent STRIPS planning, one estimating the distance of a state to the goal, and one collecting a set of promising actions. Both are based on an extension of the heuristic that is used in the HSP system. We showed how these heuristics can be used in a greedy planning algorithm, and we proved the algorithms completeness on the class of deadlock-free domains. We collected empirical evidence that the resulting planning system is among the fastest planners in existence nowadays, clearly outperforming the other state-of-the-art planners on quite a range of domains, like the *gripper*, *logistics*, *manhattan* and *tyreworld* problems.

To the author, the most exciting question is this: *Why* is the heuristic information obtained in this simple manner so good ? It is not really difficult to construct abstract examples where the greedy approach produces arbitrarily bad plans, or uses arbitrarily much time, so why does it almost never go wrong on the benchmark problems ? Why is the relaxed solution always so close to a real solution, except for the *Tower of Hanoi* problems ? Is it possible to define a notion of "natural" planning domains, where relaxed solutions have desirable properties ?

First steps into that direction seem to indicate that, in fact, there might be some underlying theory in that sense. In particular, it can be proven that the Enforced Hill-climbing algorithm, when using a heuristic that always returns an *optimal* relaxed solution, finds optimal solutions for certain planning problems, like those from the *gripper* domain. We have not yet, however, been able to identify some general structural property that implies optimality of the Enforced Hill-climbing algorithm. Moreover, there seems to be some difficulty in turning the notion "close to a real solution" to a formal definition.

A more "close to ground" piece of further work is to finetune the heuristics. For example, while the set H in most cases contains the actions that are really helpful in the next step, it also often contains actions that should be postponed to later stages in the planning process. It does not seem difficult to find these actions, or at least supply a heuristic to sort them out. Also, heuristics for selecting actions during relaxed plan extraction could be defined. For example, it might make sense to always pick the action with the easiest to achieve preconditions (sum up the levels of all preconditions).

We have seen that often, the relaxed solutions are real close to a solution plan. Sometimes, the relaxed plan *already is* an optimal plan for the current situation. Thus, one might be tempted trying to use the relaxed plans in an even more radical way than by performing greedy search. Specifically, one might try to modify a relaxed plan such that it turns into a real plan. In [Srivastava and Kambhampati, 1999], an approach has been developed which aims at turning plans that ignore resources into real plans. Now, it is characteristic for relaxed plans that they *do* ignore resources (Here, resources mean something like a robot hand in the *Blocksworld*). Thus, a combination of this approach with the method shown here might turn out to be an efficient planning system.

Finally, it remains to extend the approach to more expressive planning languages than STRIPS, which is our main objective for future work. In particular, we want to extend the FAST-FORWARD system to handle full scale ADL domains, which should not be much of a problem, and to handle resource constrained planning domains, which might be a problem. For domains containing continuous resources, we suspect that the greedy search algorithm will be inadequate. In particular, such domains will usually not be invertible: after executing a *move* action that uses fuel, one can not get back anymore, as the fuel remains used, no matter what we do next. To handle resources, we therefore have a more open search algorithm in mind, like A^* or something similar. It is not at all clear, however, wether as good results can be obtained this way. But it sure is worth investigating.

References

[Blum and Furst, 1997] Blum, A. and Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):279–298.

[Bonet and Geffner, 1999] Bonet, B. and Geffner, H. (1999). Planning as heuristic search: New results. In *Proceedings of the 5th European Conference on Planning*, pages 359–371.

[Bonet et al., 1997] Bonet, B., Loerincs, G., and Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence*, pages 714–719.

[Bylander, 1992] Bylander, T. (1992). Complexity results for serial decomposability. In *Proceedings of the 10th National Conference of the American Association for Artificial Intelligence*, pages 729–734.

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *AI*, 2:189–208.

[Fox and Long, 1998] Fox, M. and Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9.

[Kautz and Selman, 1996] Kautz, H. and Selman, B. (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence*, pages 1194–1201.

[Kautz and Selman, 1999] Kautz, H. and Selman, B. (1999). Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 318–325.

[Koehler and Hoffmann, 2000] Koehler, J. and Hoffmann, J. (2000). On reasonable and forced goal orderings and their use in an incremental planning algorithm. Submitted for Publication in JAIR.

[Koehler et al., 1997] Koehler, J., Nebel, B., Hoffmann, J., and Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *Proceedings of the 4th European Conference on Planning*, pages 273–285.

[Long and Fox, 1999] Long, D. and Fox, M. (1999). Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research*, 10:87–115.

[McDermott, 1996] McDermott, D. (1996). A heuristic estimator for means-ends analysis in planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*, pages 142–149.

[Nebel et al., 1997] Nebel, B., Dimopoulos, Y., and Koehler, J. (1997). Ignoring irrelevant facts and operators in plan generation. In *Proceedings of the 4th European Conference on Planning*, pages 338–350.

[Refanidis and Vlahavas, 1999] Refanidis, I. and Vlahavas, I. (1999). Grt: A domain independent heuristic for strips worlds based on greedy regression tables. In *Proceedings of the 5th European Conference on Planning*, pages 346–358.

[Srivastava and Kambhampati, 1999] Srivastava, B. and Kambhampati, S. (1999). Scaling up planning by teasing out resource scheduling. In *Proceedings of the 5th European Conference on Planning*, pages 174–187.