Improving Relaxed-Plan-Based Heuristics

Jorge A. Baier
Department of Computer Science
University of Toronto
Toronto, ON M5S 3G4, Canada

Abstract

Relaxed-plan-based (RPB) heuristics were first proposed by Hoffmann and Nebel for their FF system and are still used by current top-performing planners. Their main characteristic is that they are computed by computing a so-called relaxed plan, which is a plan for a relaxed version of the problem that ignores negative effects of actions. However, still in some domains that humans consider simple, they provide bad guidance. Arguably, the reason is that disregarding deletes oversimplifies those domains. Consequently, relaxed plans ignore key parts of the domain's structure. This paper describes preliminary work that attempts to identify how it is possible to compute better relaxed plans that will better respect the structure of the original (un-relaxed) problem. To that end, we propose two techniques for extracting improved relaxed plans. The first (domain-independent) technique identifies missing actions that would have to be performed if the relaxed plan was to be executed in the real (un-relaxed) world. The second (domain-dependent) technique uses domain knowledge, in the form of simple state constraints, to attempt to extract a relaxed plan that respects key information of the domain. We prove that the first technique can significantly improve the performance and quality of solutions obtained with a vanilla RPB heuristic and enforced hill climbing on a family of simple blocks-world problems. We experimentally show that both techniques improve search efficiency in example domains.

Introduction

Relaxed-Plan-Based (RPB) heuristics are among the most successful in classical planning, being key to systems such as FF (Hoffmann & Nebel 2001), and SGPlan (Chen, Wah, & Hsu 2006). To compute an RPB heuristic one constructs a plan that solves a relaxed version of the original problem. Here we focus on RPB heuristics that are computed from relaxed versions of the problem where negative effects of actions (a.k.a. *deletes*) are ignored, like the one used by FF.

RPB heuristics have been successful in planning mainly for two reasons. First, in many domains they give a very accurate advice of what needs to be done to achieve the goal in a particular state. Second, they can be computed efficiently. Finally, in many domains (e.g., *briefcase*, *logistics*) RPB heuristics are provably effective (Hoffmann 2005), since they have no local minima.

Despite their success, in some domains that are simple for humans, RPB heuristics are not effective (e.g., *blocksworld*). Ignoring deletes does not seem to be a good idea in those cases. More precisely, by relaxing deletes to compute the heuristic, we lose a key part of the underlying "structure" of those domains.

This paper describes work in progress that attempts to determine how it is possible to find improved relaxed plans/heuristics. These improved plans are obtained by changing the relaxed plan extraction phase in such a way that the resulting plan will take into account some elements that would have otherwise been ignored by a vanilla RPB heuristic. The improved relaxed plans are still efficiently computable.

We propose two techniques to compute improved relaxed plans. Both are based on the fact that relaxed plans could be considered as *advice* regarding what has to be done at each state of the search. The first is a domain-independent technique. It will modify a standard relaxed plan P when it realizes that preconditions of some actions in P will be invalidated (*occluded*) by some actions in P *no matter how* we execute P in the un-relaxed domain. When preconditions are occluded, we assume they have to be fixed by *some* action, and the heuristic is modified accordingly. The modified RPB heuristic is then proven more effective than the vanilla version in an example domain.

Arguably domain-specific knowledge—logical assertions that are entailed by the domain description—represents part of the structure of a domain. Our second (domain-dependent) technique proposes to use domain knowledge, in the form of state constraints, to inform the relaxed plan extraction phase of key parts of the structure that we would like to be preserved by the relaxed plan. We propose the use of very simple constraints that can be extracted by existing systems. We show in an example domain that our technique can improve search effectiveness.

Background

Relaxed-Plan-Based Heuristics To compute the RPB heuristic for a state s, we expand a so-called *relaxed planning graph* (Hoffmann & Nebel 2001) from s, which is no different from the graph that would be expanded by Graphplan (Blum & Furst 1997) on the relaxed instance of the problem. We view this graph as composed of *relaxed states*. A relaxed state at depth $n+1$ is generated by *adding* all the

effects of actions that can be performed in the relaxed state of depth n, and then by copying all facts that appear in layer n. The graph is expanded until the goal or a fixed point is reached.

After the graph is computed, the relaxed plan is extracted by regressing from the goal to the initial state. A *relaxed plan* can be represented as a sequence of sets of actions $A_1 \cdots A_n$. In general, many relaxed plans can be extracted from a single graph; indeed, although finding the optimal relaxed plan is NP-complete (Hoffmann & Nebel 2001), heuristics can be used to extract a good (small) one in polynomial time. The number of actions in the relaxed plan is used as the heuristic value of the node s.

If $A_1 \cdots A_n$ is a relaxed plan, then the actions in A_1 are often referred to as *helpful actions*. These actions are used by planners like FF as advice regarding the successors that most likely will lead to satisfying the goal. If h is an RPB heuristic, we denote by $\mathcal{H}(h, s)$ the set of helpful actions of the relaxed plan constructed by the algorithm for h on state s. Moreover, we use $h^*(s)$ to denote the cost of an *optimal* plan that achieves the goal starting from state s.

Planning with RPB Heuristics As with any other domain-independent heuristic for planning, a standard best-first search can be used for planning with RPB heuristics. However, the most effective planners improve over best-first search by exploiting helpful actions in one way or another. One of the reasons for this is that computing the heuristic for a state is computationally expensive. FF, for example, uses an enforced hill climbing (EHC) search algorithm that will use only helpful actions in a breadth-first search that looks for an improved successor of the node being expanded.

Concentrating on helpful actions improves the efficiency of planners but focusing on them too much can lead, on its own, to sacrificing completeness, independent of the search strategy being used.

Algorithm 1 is a planning function that exploits RPB heuristics. It is a variation of the one used by Fast Downward (Helmert 2006). The algorithm uses two independent open lists to keep the nodes in the search frontier: the $PriOpenList$ and the $SecOpenList$. The $PriOpenList$ contains nodes that were generated by performing a helpful action, and $SecOpenList$ contains those that were not. An important observation is that the computation of the heuristic is *deferred* for successors that are not generated by a helpful action; these successors therefore inherit the heuristic value of their immediate ancestor (line 19). This enables this algorithm to save time by not computing the heuristic for states that don't look promising. At the end of the while loop (line 21), the $CurOpenList$ points to the other open list (i.e., if it was originally pointing to $PriOpenList$, it will point to $SecOpenList$, and vice versa). The switch is only made if afterwards $CurOpenList$ points to a non-empty list.

Although Algorithm 1 concentrates on helpful actions, it is easy to see that it is complete for the case of classical planning, because the search space is finite. It is easy alsto to see that removing line 21 makes it incomplete, but probably faster on the instances that can be solved by only considering helpful actions.

Algorithm 1 A complete RPB-based planning algorithm.

```
 1: function RPB-PLAN(state init, goal goal)
 2:     Compute heuristic & helpful actions for init
 3:     Insert init into PriOpenList
 4:     SecOpenList ← []
 5:     CurOpenList ← PriOpenList
 6:     while PriOpenList and SecOpenList are not empty do
 7:         father ← best state in CurOpenList
 8:         Insert father into ClosedList
 9:         if father satisfies goal then
10:             return father
11:         if CurOpenList =SecOpenList then
12:             Compute heuristic & helpful actions for father
13:         succ ← successors of father
14:         for all s ∈ succ ∖ ClosedList do
15:             if action that produced s is in father.helpful then
16:                 Compute heuristic & helpful actions for s
17:                 Insert s into PriOpenList
18:             else
19:                 s.heuristic ← father.heuristic
20:                 Insert s into SecOpenList
21:         Switch CurOpenList              ▷ change to the other list
22:     return failure
```

Good Relaxed Plan Heuristics

What makes an RPB heuristic a good heuristic? A uncontentious informal answer to this question may be as follows. A good RPB heuristic is one that leads the search to a reasonably good solution, reasonably fast. Unfortunately this refers to two fuzzy concepts, whose definitions are themselves open to debate, i.e., what is "reasonably good", and what is "reasonably fast".

Rather than proposing a definition for these fuzzy concepts, in the remainder of this section we focus on two properties of RPB heuristics: *helpfulness* and *accuracy*. We justify, in intuitive terms, why these properties are important for obtaining good-quality solutions fast. (A more thorough theoretical analysis is left as future work.)

As was previously highlighted, Algorithm 1 prioritizes those successors of a state s that result from performing a helpful action in s, by computing their heuristic value immediately after being expanded. Intuitively, we would like to prioritize successors s that lie on an optimal plan from s.

Definition 1 (Helpful heuristic function[1]). *Let $Opt(s)$ denote the set of optimal plans that lead from s to a goal state. An RPB heuristic h is* helpful *in state s if $\mathcal{H}(h, s)$ contains an action a that is a prefix of a plan in $Opt(s)$.*

Figure 1 depicts a situation which shows why helpful heuristics can be better than un-helpful ones. In the picture, s and s' are respectively in the primary and secondary

[1] Hoffmann (2005) has considered a similar concept that refers to the so-called actions *respected by the relaxation*. If all actions in the problem are respected by the relaxation, the RPB heuristic is helpful, but not vice versa.

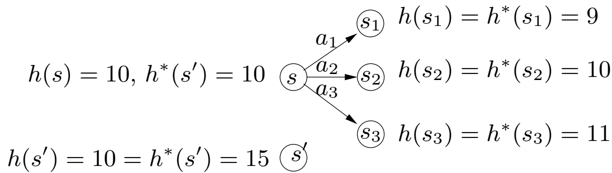


Figure 1: States s_1, s_2, and s_3 are generated by respectively performing a_1, a_2, and a_3 on s.

open lists, and s is being expanded. The expansion of s produces three successors: s_1, s_2, and s_3. Consider the case where a_1 is a helpful action. Then, s_1 (the closest state to the goal) will be the next state in the primary open list to be expanded after s. Consider the case where a_3 is helpful but a_1 is not helpful (i.e., the heuristic is not helpful in s). In this scenario, s_1 is put in the secondary open list, and will be associated with the heuristic value of its father (10). The next state to be expanded will be s (because it is a shorter plan in the secondary open list). On the next iteration, s_4 will be expanded. Finally, in the next iteration s_1 *may be* expanded, since it will have the same heuristic value as *all* other successors of s' that are not helpful. Though s_1 might eventually be expanded, it will certainly not be as quickly as we would ideally want.

Helpfulness, on its own, guarantees that an otherwise incomplete version of Algorithm 1 is complete:

Proposition 1. *If a helpful heuristic is used, Algorithm 1 is complete even if line 21 is removed.*

As also remarked by Helmert (2006), the previous result is good news in terms of efficiency, since we can avoid computing the heuristic for many states in the search space.

Helpfulness is still not enough to guarantee that the planner will be heading in the right direction. The reason is that the heuristic *value* of a state is the actual criterion for picking up nodes from the open lists. If we want the heuristic to "pick the right node," the heuristic value of a node s should be related to the actual optimal cost $h^*(s)$ of a plan from s. In particular, if from two successors of a node, say s_1 and s_2, it is the case that $h^*(s_1) < h^*(s_2)$, then we would never want $h(s_1) > h(s_2)$ to hold true. We say that a heuristic function is *accurate for siblings*, or simply accurate, when it completely respects the ordering of successor nodes in terms of their actual optimal cost to reach the goal. More formally,

Definition 2 (Accurate heuristic function)**.** *A heuristic function h is accurate for state s if for any two successors of s, s_1 and s_2, $h^*(s_1) < h^*(s_2)$ iff $h(s_1) < h(s_2)$. A heuristic is accurate for planning problem P, if it is accurate for every s in the search space.*

Note that accuracy *does not* imply admissibility, as it refers to the relative heuristic values of sibling nodes, not to arbitrary nodes from the search space.

Helpfulness and accuracy are desirable properties of heuristics, however it is hard to obtain them in practice. We think it might be impossible to prove that any given RPB heuristic has any of these properties, independently of the domain. However, thinking about these properties is useful to improve the quality of existing RPB heuristics. We shall see this in the following sections.

Occlusion Penalties

As we remarked earlier, an accurate heuristic function enables a planner such as the one in Alg. 1 to pick the right successor of a node during planning. In this section we show a very simple planning domain in which RPB heuristics are *not* accurate. We then define the notion of occlusion, which is a way of considering negative effects of actions in RPB heuristics. Using this notion, we can transform a given RPB heuristic that is not accurate into an accurate one.

An Example in the Blocks World

Let us consider the old and well-known blocks world, with the standard operators $stack$, $unstack$, $pickup$, and $putdown$. More specifically, let's consider the family of very simple problems that was suggested in the call for papers for this workshop, i.e., building a sorted tower with blocks $B_1, B_2 \ldots, B_n$, with B_1 on top. Initially, all blocks are lying on the table, except for B_n, which is on B_1. Amazingly, the FF planner needs more than 1GB of memory when $n \geq 28$. Moreover, on the instances that it can solve, it gives very poor solutions. For example, the plan for instance with $n = 27$ contains 103 actions, almost double the number in the optimal, 54-action plan. SGPlan, on the other hand, returns a 91-action plan.

Although both FF and SGPlan rely on RPB heuristics, the low quality of their solutions can be explained mostly by the particular search strategy employed rather than the heuristic. Nevertheless, vanilla RPB heuristics are still not good for this family of problems. Actually, even algorithms like Algorithm 1 will not find good solutions.

To understand why this happens, consider the domain with $n = 3$ and an intermediate state where all blocks lie on the table. The successors to this state are those that come from performing $pickup(B_i)$, for $i \in \{1, 2, 3\}$. Naturally, the optimal decision here is to perform $pickup(B_2)$, to then stack it over B_3. However, a vanilla RPB heuristic will not suggest that. Indeed, the optimal relaxed plan for the successor where $pickup(B_1)$ is performed is as follows:

$$\{stack(B_1, B_2)\}\{pickup(B_2)\}\{stack(B_2, B_3)\}, \quad (1)$$

and the optimal relaxed plan for $pickup(B_2)$ successor has the same number of actions:

$$\{stack(B_2, B_3)\}\{pickup(B_1)\}\{stack(B_1, B_2)\}. \quad (2)$$

By ignoring negative effects, the RPB heuristic makes the planner behave as if these two successors were identical. Rephrasing this in the terminology of the previous section, the standard RPB heuristic is not accurate.

Making an RPB Heuristic Accurate

The reason why the RPB heuristic is inaccurate for this family of blocks-world examples is because negative effects are ignored. If we took both relaxed plans seriously, and wanted to execute them in the un-relaxed domain, we would only be able to perform plan (2). The problem of (1) is that

action $stack(B_1, B_2)$ deletes or *occludes* a precondition of $pickup(B_2)$. Below we define formally the concept of occlusion but first we need another definition.

Definition 3. *Let P be a relaxed plan constructed for a state s, and let a and b be actions in P. Moreover, let P_{-a} denote the plan that corresponds to deleting action a from P. We say that a* is necessary to *b—denoted by $a \prec b$—if the execution of b is not possible while attempting to perform P_{-a} in the relaxed domain.*

Intuitively, $a \prec b$ expresses that a is essential to achieving the preconditions of b. That is, $a \prec b$ implies that necessarily, in any valid linearization of the plan, a has to occur before b.

Definition 4 (Occluded facts)**.** *A fact f in a relaxed plan is* occluded *if f is a precondition of an action b in P such that (1) there exists an action a in P such that $a \prec b$, and a deletes fact f, and (2) every action c in P that* adds *f is such that $c \prec a$ or $b \prec c$.*

A precondition f of an action a is only occluded when an action a' that is needed by a has deleted it. However, other actions in the relaxed plan (referred to as b in the definition) could have added f, and therefore un-occlude f. The definition states that almost all actions that add f can un-occlude f *except* those that necessarily have to occur before a' or after a.

Note that under the definitions above, the precondition $clear(B_2)$ of $pickup(B_2)$ is occluded in plan (1) by action $stack(B_1, B_2)$. Note that the only action that adds $clear(B_2)$ in (1) is $stack(B_2, B_3)$ but this action cannot un-occlude the fact because $pickup(B_2) \prec stack(B_2, B_3)$.

The definition of occlusion can also be extended to goal facts. A goal fact is occluded if it is deleted by an action that occurs after the one that added it.

Now we focus on how we can use the notion of occlusion to repair an estimate given by a RPB heuristic. In a nutshell, if an action's precondition is occluded, there is no valid linearization of the plan that will reach the goal in the un-relaxed domain. Under the premise that the relaxed plan is still a good approximation to a solution, we could attempt to repair it, or attempt to estimate how many more actions it would need to become a "real" plan. From the two alternatives, we have experimented with the latter.

Algorithm 2 computes *occlusion penalties*. The occlusion penalty is a number that estimates the number of actions that are missing in the relaxed plan. It does so by assuming that each occluded fact will be un-occluded by some action that adds this fact. Whenever an occluded fact f is found, a pseudo-action Add_f is added to the relaxed plan right after the occluding action. The only effect of Add_f is to add f. Note that the addition of Add_f cannot introduce additional occlusions.

Now instead of using the length of the relaxed plan as a heuristic value for a node, we can use the length of the original plan plus the occlusion penalty. In our example this would mean that a successor resulting from $pickup(B_1)$ (the wrong action) gets a penalty of 1, and therefore its heuristic value is 4, rather than 3. This modified heuristic is accurate for this family of blocks-world.

Algorithm 2 An algorithm for occlusion penalties

```
1: function OCCLUSIONPENALTY(relaxed plan P = A_1 ··· A_n)
2:     penalty ← 0
3:     while Exists fact f occluded by action a do
4:         Let i be such that a ∈ A_i
5:         A_{i+1} ← A_{i+1} ∪ {Add_f}
6:         penalty ← penalty + 1
7:     return penalty
```

Moreover, extending an RPB heurisitc with occlusion can lead to polynomial performance on this family of blocks world problems. Let h^+ be the RPB introduced by Hoffmann (2005) that computes the optimal relaxed plan for any given state. Let h^+_o be h^+ extended with occlusion penalties. Then, we obtain the following result.

Proposition 2. *Let $s_0 \cdots s_{2n}$ be the state path traversed by the optimal plan that solves the blocks-world problem for n blocks defined earlier in this section. Moreover, let $HSucc(s)$ denote the set of successors of s that are produced by some action in $\mathcal{H}(h^+_o, s)$. Then for any $i \in \{0, \ldots, 2n-1\}$, $s_{i+1} \in HSucc(s_i)$, and for any $s' \in HSucc(s_i) \setminus \{s_{i+1}\}$, $h^+_o(s_{i+1}) < h^+_o(s')$.*

This property—which is not true of h^+—intuitively says that h^+_o will lead any search algorithm focusing on helpful actions to *always* choose the optimal node to be expanded next. In particular, this implies that EHC solves this family of problems optimally in polynomial time in n.

Incorporating Domain Knowledge

As we saw earlier, helpful heuristics are desirable because they can speed up search. However RPB heuristics sometimes lack this property because relaxed plans ignore some of the structure of the problem. To address this issue, in this section we propose the use of *domain knowledge* to extract better (helpful) relaxed plans.

In contrast to some previous work in planning, which has used non-trivial domain knowledge to improve search (e.g., LTL formulae to control search by Bacchus & Kabanza (1998) using TLPLAN), we use much simpler knowledge to improve relaxed plan generation. Indeed, we will just use state constraints, i.e., properties that hold true in every state of the plan. In the rest of the section we show a very simple domain in which RPB heuristics are not helpful. Then we show how domain knowledge can be useful for extracting better relaxed plans, and end giving an algorithm for extracting relaxed plans that *tries* to satisfy given constraints.

An Example in the Storage Domain

The storage domain was introduced in the 2006 International Planning Competition (IPC-5) (Gerevini *et al.* 2006). In the STRIPS version of this domain, there are *crates* that can be transported using *hoists* from one *area* to another. Crates can be *lifted* and *dropped* by hoists. Hoists can carry at most one crate at a time, and can move between connected areas. A hoist can only move to *clear* areas, and can pick up or

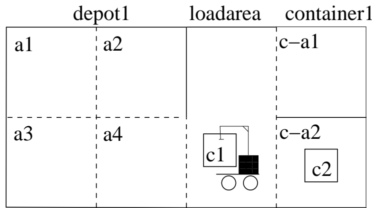


Figure 2: An intermediate state in a 2-crate storage domain. Dashed lines divide connected areas; e.g. $a4$ is connected to $loadarea$ but $a2$ is not. The goal is to have all crates in depot1. The hoist is currently holding $c1$.

drop a crate in an area that is connected to the one they are currently in. A crate can only be dropped in a clear area. Areas can be part of a depot, part of a container, or just be used to move between depots and containers.

Usually problems in this domain consist of moving crates located initially in containers to the depots. These problems are quite simple for humans, however, they turned out to be surprisingly hard for most of the IPC-5 competitors. For example, no planner, except SGPlan, is able to solve instances 19-30. If one looks at instance 19 and over, there is no significant additional number of objects—which would be an obvious reason to justify bad performance.

Vanilla RPB heuristics have some serious drawbacks on this domain. Consider the situation depicted in Figure 2. A relaxed plan for this state would be as follows.

$$\{drop(c1, a4)\}\{pick(c2, c\text{-}a2)\}\{drop(c2, a4)\} \quad (3)$$

The drawback of this relaxed plan is that it is *not being helpful* because is does not have the action $go\text{-}in(a4)$ in the first layer. This relaxed plan is suggesting that a good thing to do is to drop $c1$ immediately, a clearly dumb decision.

The source of the problem is, again, in the overly relaxed domain that is being solved. Relaxing deletes completely is a bad idea because it allows the relaxed plan to consider—among other things—dropping crates at the same location, ultimately causing it not to consider the *go-in* action.

A potential fix to the problem would be to define a less relaxed problem, in which certain fluents are *not* relaxed. This would mean that deletes would *have* to be considered during planning extraction. In our example, the fact that we are relaxing the *clear* fluent looks particularly harmful, and we might want to un-relax it. This approach however is not good in theory, since extraction gets worst-case exponential in the depth of the graph. Moreover, in practice it doesn't seem to be promising either. Actually, SatPlan (Kautz, Selman, & Hoffmann 2006), takes more than 20 sec. on a 2-GHz Pentium to solve problem number 21 of the storage IPC-5 problem set when only the *clear* predicate is un-relaxed.

Using Domain Knowledge to Achieve Helpfulness

Domain knowledge has been previously used to improve search efficiency in planning. Most previous work has concentrated on using this knowledge to control or prune the search space, as in TLPLAN (Bacchus & Kabanza 1998). Here we propose to use domain knowledge for relaxed plan extraction.

The type of domain knowledge we propose to use is among the simplest one can think of: domain state constraints. These constraints are logical formulae that hold in every state of the search space. For example, in the storage domain *"two objects cannot be at the same store area"*, and *"the hoist can lift only one crate at a time"* are properties that hold in every problem. Existing systems are able to extract these constraints. Indeed, DISCOPLAN (Gerevini & Schubert 2000) is able to extract both of the constraints mentioned above.

Relaxed plan extraction that respects an arbitrary state constraint is worst-case exponential in the depth of the plan. The intuitive reason is that state constraints may imply that deletes must not be ignored.

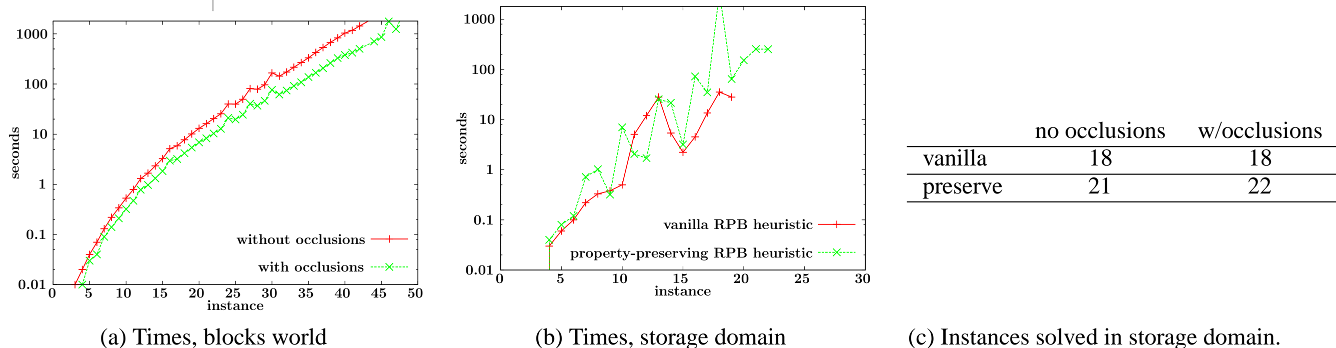
Nevertheless, we do not need to *fully* satisfy domain constraints to obtain reasonable relaxed plans. Sometimes a relaxed plan that "almost" satisfies a state constraint is much better than one that doesn't.

Algorithm 3 extracts relaxed plans while trying to achieve a domain constraint as much as possible. The main difference between ours and a standard relaxed plan extraction algorithm is in lines 5-10. Instead of looking for any action that achieves an unsatisfied sub-goal, the algorithm looks for an action such that the union of its effects plus the effects of all other actions already in the relaxed plan satisfy the constraint. If such an action exists, it is added to the relaxed plan in the standard way. Otherwise the algorithm *does not* add any action to the relaxed plan but still length is incremented by one (line 10) to reflect the fact that at least one action was needed to satisfy the unsatisfied sub-goal. The returned length is used by the search algorithm as the heuristic value of the node being evaluated.

Algorithm 3 Property-preserving relaxed plan extraction

```
1:  function ExtractPrsrv(plan graph P_0 A_0 P_1 ··· A_{n-1} P_n,
    goal G, property φ)
2:      P_n^rel ← G                                  ▷ initialize goals
3:      for i = n ... 1 do
4:          for all p ∈ P_i^rel do                   ▷ find supporting actions
5:              Find a ∈ A_j such that j < i, p ∈ eff+(a), and
6:                  such that φ is satisfied in ∪_{i=0}^n P_i^rel ∪ eff+(a)
7:              if a is found then
8:                  A_j^rel ← A_j^rel ∪ a
9:                  P_j^rel ← P_j^rel ∪ prec(a)
10:             length ← length + 1                  ▷ Count action
11:     return ⟨length, A_0^rel ··· A_n^rel⟩
```

Algorithm 3 is polynomial in the size of the relaxed plan. In fact, compared to the standard relaxed plan extraction, the only extra cost is finding an action that preserves φ. Efficiency however comes at a cost: the algorithm is not complete; it will not find a relaxed plan that respects φ for an arbitrary φ, if one exists. The main reason for this is that the algorithm cannot backtrack: once an action a has been included in the relaxed plan, it can never be removed.



(a) Times, blocks world

(b) Times, storage domain

	no occlusions	w/occlusions
vanilla	18	18
preserve	21	22

(c) Instances solved in storage domain.

Figure 3: Summary of experimental results

An important observation is that the algorithm in a way "gives up" when an action violates a property by not adding any action to the relaxed plan. This is useful for properties that, once violated, will never be true again. By not committing to any action, there is still a chance of finding other actions later that will still satisfy the property.

As a final remark, the quality of the relaxed plans found by Alg. 3 depends strongly on the planning graph that is given to it as input. Generally, a plan that satisfies a state constraint is *longer* (in terms of makespan) than one that does not satisfy it. Therefore, we must feed our algorithm with planning graphs that are extended beyond the layer where the goals appear for the first time.

Let us go back to our example domain. Our initial intuition is that the reason why the vanilla RPB heuristic is bad is that it allows locating an arbitrary number of crates in the same position. Suppose we consider the following state constraint:

```
(forall c1 c2 - crate
   (forall a - area
      (implies (and (on c1 a) (on c2 a))
               (= c1 c2))))
```

then Algorithm 3 can find the following relaxed plan:

$\{drop(c1, a4), go\text{-}in(a4)\}\{pick(c2, c\text{-}a2)\}\{drop(c2, a3)\}$.

As before, action $drop(c1, a4)$ achieves the first subgoal $in(c1, depot1)$. To achieve the second subgoal $in(c2, depot1)$, the algorithm chooses $drop(c2, a3)$, since choosing $drop(c2, a4)$ would violate the constraint. The remaining actions are added to satisfy the preconditions of $drop(c2, a3)$. The resulting heuristic is helpful for the state of Figure 2.

Implementation & Preliminary Experiments

We have implemented (deterministic) versions of the algorithms given above for computing occlusion penalties and property-preserving relaxed plans. Our planner is a modified version of TLPlan which uses a domain-independent heuristic search algorithm similar to Alg. 1.

We have performed a preliminary evaluation of our techniques in the blocks family of problems introduced above, and on the IPC-5 storage problems. In blocks world we found that occlusion penalties were key to finding good-quality solutions relatively fast. As shown in Fig. 3(a) the version with occlusions is faster and solves more problems than the version without occlusions. Moreover, occlusions enable the planner to solve *all* problems optimally. The vanilla RPB, on the other hand, yields plans with 4 more actions for almost all instances.

We have also experimented with our property-preserving algorithm in the storage domain. Fig. 3(c) shows a summary of the number of instances solved given a 1 hour timeout. The property used is the one that restricts having two crates in the same location. Not surprisingly, property-preserving heuristics are sometimes slower; however, in general, times are comparable (see Fig. 3(b)).

Discussion & Related Work

We have presented two techniques for improving relaxed plans by attempting to incorporate structure of planning domains that is lost by the standard relaxation of negative effects. Our first technique is domain independent, and could be understood as a way of considering some mutexes during relaxed plan extraction. We have proven that this techinque can significantly improve the performance of EHC in a family of simple blocks-world problems.

The second, domain-dependent technique, could be understood as a way of customizing relaxation, but also as a way of considering mutexes during relaxed plan extraction. The relationship between our two techniques still has to be established. We think that the incorporation of domain knowledge in relaxed plan extraction has a great deal of potential, especially if we could mechanize the selection of useful state constraints to be considered. For example, in the storage domain, it does not seem to be reasonable to consider the constraint where only one crate is allowed to be carried by a hoist.

There are pieces of work in the literature that are related to this work. First, work that incorporates mutexes for heuristics not based on relaxed plans, such as that by Nguyen & Kambhampati (2000), and work that extracts better relaxed plans for cost-based planning (e.g. (Do & Kambhampati 2003)). More related is work that exploits generic types to improve relaxed plans by Coles & Smith (2006). Their

work uses mechanically extracted *type-knowledge* to recognize when actions need to be added to the relaxed plan. Specialized techniques, that depend on the particular type of objects, are used to improve the plans.

Acknowledgments

I am very grateful to Sheila McIlraith for useful discussions and comments on drafts of this paper. I thank the anonymous reviewers for their suggestions and insightful comments.

References

Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence* 22(1-2):5–27.

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

Chen, Y.; Wah, B.; and Hsu, C.-W. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research* 26:323–269.

Coles, A., and Smith, A. 2006. Generic types and their use in improving the quality of search heuristics. In *Proceedings of the 25th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2006)*.

Do, M. B., and Kambhampati, S. 2003. Sapa: A scalable multi-objective metric temporal planner. *Journal of Artificial Intelligence Research* 20:155–194.

Gerevini, A., and Schubert, L. K. 2000. Discovering state constraints in DISCOPLAN: Some new results. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-00)*, 761–767.

Gerevini, A.; Dimopoulos, Y.; Haslum, P.; and Saetti, A. 2006. 5th International Planning Competition. `http://zeus.ing.unibs.it/ipc-5/`.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

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

Hoffmann, J. 2005. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research* 24:685–758.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SatPlan: Planning as satisfiability. In *5th International Planning Competition Booklet (IPC-2006)*.

Nguyen, X., and Kambhampati, S. 2000. Extracting effective and admissible state space heuristics from the planning graph. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-00)*, 798–805.