Improving Planning Performance Using Low-Conflict Relaxed Plans

Jorge A. Baier
Department of Computer Science
University of Toronto
Canada

Adi Botea
NICTA and
the Australian National University
Canberra, ACT

Abstract

The FF relaxed plan heuristic is one of the most effective techniques in domain-independent satisficing planning and is used by many state-of-the-art heuristic-search planners. However, it may sometimes provide quite inaccurate information, since its relaxation strategy, which ignores the delete effects of actions, may oversimplify a problem's structure. In this paper, we propose a novel algorithm for computing relaxed plans which – although still relaxed – aim at respecting much of the structure of the original problem. We accomplish this by generating relaxed plans with a reduced number of *conflicts*. An action a will add a conflict when added to a relaxed plan if the resulting plan is provably illegal (i.e, not executable) in the un-relaxed problem. As a second contribution, we propose a new lookahead strategy, in the spirit of Vidal's YAHSP lookahead, that can better exploit the contents of relaxed plans. In our experimental analysis, we show that the resulting heuristic improves over the FF heuristic in a number of domains, most notably when lookahead is enabled. Moreover, the resulting system, which uses our new lookahead, is competitive with state-of-the-art planners, and even better in terms of the number of solved problems.

Introduction

In domain-independent satisficing planning with heuristic state-space search, the quality of the heuristic has a strong impact on the planner performance. Some of the most successful approaches to compute heuristics are based on relaxations which ignore parts of the interactions between actions. The *relaxed plan* (RP) heuristic, introduced in the FF planner (Hoffmann and Nebel 2001), estimates the actual length of a plan by the length of a plan for a relaxed instance. It is usually quite effective; indeed, it is used by many state-of-the-art planners, including several systems awarded in recent planning competitions.

In addition to using the RP length as a heuristic, the actual contents of RPs can be further exploited by identifying helpful actions (Hoffmann and Nebel 2001), building lookahead policies (Vidal 2004) and building macro-operators (Botea, Müller, and Schaeffer 2007). These techniques have been shown to be very effective for many planning benchmark problems. As they exploit the contents of RPs, their success depends directly on the similarity between the relaxed and the actual plans.

However, the RPs obtained by the method of Hoffmann and Nebel can dramatically differ from actual plans in some respects. For example, the RP never changes any fact more than once, even when the corresponding actual plan may involve making some facts true and false several times repeatedly. Moreover, an FF-style RP never contains any action more than once, even though the actual plan could use an action several times.[1] For n actions, the maximum obtainable plan length is n, even though in some cases the actual plan might require a greater number of actions (which could be even exponential in n).

In this work we consider methods that aim at computing RPs that are more accurate, i.e., more similar to real plans. In addition, to better exploit not only the length but also the contents of a relaxed plan, this paper presents a new algorithm to construct lookahead policies, that can be viewed as an extension of Vidal's technique.

To improve the accuracy of an RP, our technique attempts to explicitly minimize the number of conflicts in the RP during its construction. A conflict is introduced by an action that generates a state with a pair of facts that cannot hold both at the same time. Conflicts bear similarities with flaws in partial-order planning (McAllester and Rosenblitt 1991) and permanent mutexes in Graphplan (Blum and Furst 1997). Formally, we define an optimization problem whose objective is to find an RP with a minimum number of conflicts, and propose a greedy approximation algorithm that can quickly obtain a low-conflict RP. By minimizing the number of conflicts in the RP we indirectly incorporate into it more of the structure that is lost when completely ignoring the actions' delete effects.

On the other hand, our algorithm for computing lookahead policies differs from Vidal's method in that we can *insert* new actions into an RP, without having to remove an RP action instead. This allows to build longer macros, which are capable of taking larger steps towards the goal.

Experiments demonstrate that the enhancements described in this paper can lead to significant improvements in search performance. The resulting system is competi-

[1] The situation is slightly more sophisticated in domains with conditional effects. We skip the details for the sake of simplicity.

tive with state-of-the-art planners such as LAMA (Richter, Helmert, and Westphal 2008) and YAHSP (Vidal 2004), and is even better in terms of the number of problems solved.

The next section contains background information. We then formalize our approach as an optimization problem that minimizes conflicts, proceeding then to describe the details of our method for RP extraction. The presentation of the new lookahead technique follows. Finally, we include an experimental evaluation, followed by a discussion.

Background

A STRIPS planning problem P is a tuple $\langle \mathcal{F}, \mathcal{I}, \mathcal{O}, \mathcal{G} \rangle$, where $\mathcal{F}$ is a finite set of facts, $\mathcal{I} \subseteq \mathcal{F}$ is an initial state, $\mathcal{G} \subseteq \mathcal{F}$ are goal facts, and $\mathcal{O}$ is a finite set of action operators that map a state (i.e., a collection of facts) into another state. An operator a is specified as a triple $\langle prec(a), add(a), del(a) \rangle$ of lists of facts for, respectively, preconditions, positive effects, and negative effects (*deletes*). The facts *prevailed* by a, $prev(a)$, are those preconditions of a that are not deleted by a. We write as $\gamma(s, a)$ the state obtained by applying a to s. For a collection of actions that can be applied in a sequence, $\gamma(s, a_0 a_1 \dots a_n) = \gamma(\gamma(s, a_0), a_1 \dots a_n)$. The set $\mathcal{A}(s)$ contains all actions applicable to a state s.

The regression of a set of goals G over an action a is the minimal set of facts that have to hold true in a state to ensure the satisfaction of G after performing a. Formally, the regression of G over a is $Regr(a, G) = (G \setminus add(a)) \cup prec(a)$.[2] For a nonempty sequence of actions $a_0 \cdots a_n$ we define $Regr(a_0 a_1 \cdots a_n, G) = Regr(a_0, Regr(a_1 \cdots a_n, G))$.

P^+, the *delete-relaxation* of P, is a planning problem just like P but in which the delete lists of operators are empty. A plan for P^+ is called a *relaxed plan* (RP) for P. Formally, a sequence of actions σ is an RP for P iff $Regr(\sigma, \mathcal{G}) \subseteq \mathcal{I}$.

Using RP-based heuristics, the heuristic value assigned to a state s is the length of $RP(s)$, an RP from s to $\mathcal{G}$. The RP is computed in polynomial time by first building a (mutex-free) planning graph for P^+ until it contains all goals. Then, for each goal, an action that achieves it (*achiever*) is selected and a new set of goals is obtained by regressing the goals over the achiever. This repeats until all regressed goals hold in s. FF chooses an achiever for a subgoal g heuristically. First, it selects the *earliest* ones, i.e., the ones responsible for the first appearance of g in the planning graph. Then, among the earliest achievers, it chooses one with the cheapest preconditions. A precondition's cost is equal to the earliest graph level that contains it.

RP heuristics can be used in combination with standard search algorithms, but RPs can be exploited further. For example, while expanding a state, FF prioritizes the so called *helpful* actions, which are the applicable actions that achieve facts marked as goals on the second atom layer of the RP. Particularly relevant to this work is the use of *forward* (or *lookahead*) *macros* built from RPs (Vidal 2004; Botea, Müller, and Schaeffer 2007). A forward macro is a sequence of actions that can be applied, in the un-relaxed world, to a state s, allowing to take longer steps towards the goal in a search. Their effectiveness, however, depends directly on the similarity between an RP and a real solution.

For example, the YAHSP planner (Vidal 2004) builds a lookahead macro iteratively, by removing an action a from $RP(s)$ and appending it to the current macro prefix. When none of the remaining actions in $RP(s)$ can be appended to the current macro prefix, a plan repair strategy will, if possible, replace an action in $RP(s)$ with one or possibly more actions, suitable to be appended to the macro. The procedure traverses (the remaining part of) $RP(s)$ repeatedly, until no more actions can be appended to the macro.

A Motivating Example

We motivate our *conflict minimization* approach with an example in the *storage* domain, a benchmark 2006 International Planning Competition (IPC-2006). In the STRIPS version of this domain, there are *crates* that can be transported from one *area* to another using *hoists*. 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 a *clear* area, and can pick up or drop a crate in an area that is connected to the one it is currently in. A crate can only be dropped in a clear area. Depots and containers are composed by connected areas.

The planning tasks considered for this domain consist of moving a set of crates located initially in containers to their destination depots. As such, given an instance, both deciding whether there is a solution, and computing a non-optimal solution can be done in polynomial time in the size of the problem. Nonetheless, these problems turned out to be surprisingly hard for most of the IPC-2006 competitors: none of them but SGPlan$_5$ (Hsu et al. 2007) could solve instances 19-30, even though these instances do not show a significant increase in size. Interestingly – as we see later in our experimental evaluation – these problems are also very hard for LAMA (Richter, Helmert, and Westphal 2008), the winner of the IPC-2008 satisficing track.

Storage is hard for many heuristic-based planners because their heuristics oversimplify the problem. We illustrate this with a simple situation depicted in Figure 1. An RP, as computed by FF, for this state would be as follows:

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

where $drop(c, to, from)$, stands for dropping c in area to while the hoist is located in area $from$. This RP obviously underestimates the number of actions required to reach the goal. However, it has another problem which can be even more critical to the scale-up of state-of-the-art planners: it does not include $go\text{-}in$ (the action that makes the hoist go in the depot). Thus, planners that rely on helpful actions will never prioritize – and might not even consider – the use of the only actually useful action in this situation.

Although the focus of this paper is on RPs, it is interesting to note that other domain-independent heuristics, like the causal graph (CG) heuristic (e.g., Helmert 2006) and its recent generalization (Helmert and Geffner 2008) have similar limitations in this particular situation. Indeed, since

[2] In regression planning the regression of G with respect to a is defined only when $del(a) \cap G = \emptyset$. In the relaxation, this condition always holds, so we omit it from the definition of $Regr$.

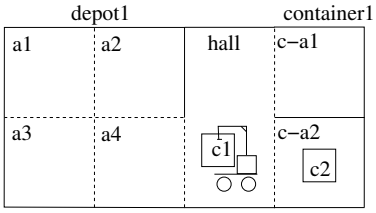

Figure 1: An intermediate state in a 2-crate storage task. Dashed lines divide connected areas; e.g. $a4$ is connected to *loadarea* but $a2$ is not. The goal is to have all crates in depot1. The hoist is currently holding $c1$.

these heuristics consider top-level goals as essentially independent, they will also behave as if all crates could be put in the same location, and never thus consider $go\text{-}in$ as part of a solution. In the context of the Fast Downward planner (Helmert 2006), this means that $go\text{-}in$ is not a "preferred operator," but in more general terms, any RPs that one would extract from a solution to the CG relaxation will potentially exhibit the same issue.

As stated earlier, our approach addresses over-relaxation by attempting to minimize the *conflicts* appearing in an RP. As such, our algorithm tries to avoid the appearance of both $drop(c1, a4)$ and $drop(c2, a4)$ in a single plan, as this generates a *delete conflict*: either action deletes $clear(a4)$ which is a precondition of the other action. As a consequence of minimizing this conflict, the algorithm considers dropping the crates in different locations. Ultimately, $go\text{-}in$ will be considered a helpful action, since it will appear in the RP because it is the only action that leads to the achievement of the precondition of a drop in a depot area other than $a4$.

Relaxed Planning as Optimization

Rather than being any solution to the delete-relaxation, we view RPs as approximate solutions to an optimization problem, which is to minimize the number of conflicts in the RP. The definition of this optimization problem is useful for two reasons: (1) it sets the foundation for the construction of low-conflict RPs, and (2) it provides an intuitive justification of the potential benefits of using low conflict RPs.

We introduce formal definitions for conflicts below. Intuitively, when a conflict appears in an RP, then the execution of the RP in the unrelaxed world will visit an illegal state. Our approach captures illegal states that are characterized by *permanent mutex sets of facts*.

Definition 1 (Permanent Mutex Set of Facts) *A set of facts $F = \{f_1, \ldots, f_n\}$ is a* permanent mutex *for a problem P if there is no state s reachable from $\mathcal{I}$ such that $F \subseteq s$.*

Example 1 $\{at(cr1, a2), at(cr2, a2)\}$ is a permanent mutex in storage, since crates $cr1$ and $cr2$ cannot both occupy the same location $a2$.

It is known that permanent mutexes may not characterize all illegal states in a planning problem. If, for example, a fact f_1 can only exist in conjunction with another fact f_2, then any state that contains f_1 but not f_2 is illegal but does not contain a permanent mutex involving f_1 or f_2. Using permanent mutexes to prove state illegality, is common practice in regression planning (see e.g., Haslum and Geffner 2000).

Sets of permanent mutexes are hard to compute, and therefore our algorithm will restrict to only *binary* permanent mutex sets, since a subset of these can be computed relatively quickly. In a preprocessing step, for the initial state only, we compute the h^2 heuristic (Haslum and Geffner 2000) which, for each pair of atoms (f_1, f_2), gives a lower bound on the distance to a state where both f_1 and f_2 are true. Pairs with an infinite distance are permanent mutexes.

Now we are ready to formally define conflicts. We consider three types of conflicts that an action could introduce when added to an RP. In the following definitions, a is an action that achieves a fact in a set of (goal) facts G (i.e., $add(a) \cap G \neq \emptyset$).

Definition 2 (Add-Prevail Conflict) *Action a produces N* add-prevail conflicts *iff there exist N permanent mutex sets formed with elements of both $prev(a) \cup add(a)$ and G.*

Definition 3 (Precondition Conflict) *Action a produces N* precondition conflicts *iff there exist N permanent mutex sets formed with elements of both $prec(a) \setminus prev(a)$ and $Regr(a, G)$.*

Definition 4 (Delete Conflict) *Action a produces N* delete conflicts *iff $|G \cap del(a)| = N$.*

An add-prevail conflict is produced when the action a that achieves a goal in G adds or maintains facts that are incompatible with some facts in G. A precondition conflict is created when some of a's preconditions are mutex with the regressed goals. A delete conflict occurs when the action just added deletes one of the subgoals we want to achieve.

The Optimization Problem Let P be a STRIPS planning problem. Consider the following optimization problem:

$$\min \text{Conflicts}(\sigma, \mathcal{G}) \text{ s.t. } \sigma \text{ is a plan for } P^+, \qquad (1)$$

where $\sigma = a_0 a_1 \cdots a_n$ is an action sequence, and $\text{Conflicts}(\sigma, \mathcal{G})$ is the sum over $i \in [0, n]$ of the number of conflicts introduced by a_i on $Regr(a_{i+1} \cdots a_n, \mathcal{G})$.

Proposition 1 $\text{Conflicts}(\sigma, \mathcal{G}) = 0$ *iff σ is a plan for P.*

Proof for $\Rightarrow$: The absence of delete conflicts implies $Regr$ coincides with standard regression in STRIPS. Hence, since $Regr(\sigma, \mathcal{G}) \subseteq \mathcal{I}$, the linear sequence σ is a plan for P. □

The proof of the proposition shows that precondition and add-prevail conflicts could be omitted from the optimization problem and we would still obtain a valid plan via minimization of only delete conflicts. However, keeping this constraint is useful to *anticipate* delete conflicts. To see this, assume we are building an RP using regression and that the current suffix is σ. Adding an action a to (the head of) σ may not introduce a delete conflict, but if it introduces any of the other conflicts, no legal plan can end with $a\sigma$, and thus *any* RP ending with $a\sigma$ has a delete conflict.

Example 2 Let $pick(c2, a2), drop(c2, a4, hall)$ be an RP suffix for the situation of Figure 1, and let G be the set of regressed goals. Note that both $at(hoist, loadarea)$ and $clear(a4)$ are in G (introduced by $drop(c2, a4, hall)$). Then $drop(c1, a4, hall)$ introduces a delete conflict in G since it deletes $clear(a4)$. Action $drop(c1, a2, a4)$ does not introduce a delete conflict, but it does introduce a prevail conflict because one of its prevailed facts, $at(hoist, a2)$, is mu-

tex with $at(hoist, loadarea)$. As shown in the next section, the latter conflict can be eliminated by inserting the $go\text{-}out(hoist)$ action in the RP.

Low-Conflict RPs via Greedy Minimization

Our objective is to improve planning performance by utilizing low-conflict (i.e., improved) relaxed plans. A low-conflict RP is one that has few conflicts in the sense of the optimization task defined in (1). This section describes an algorithm that aims at producing low-conflict RPs by greedily approximating a solution to the optimization problem. Recall that an exact solution is equivalent to finding a plan, and thus is not a reasonable approach if one wants a heuristic. This section starts by outlining the algorithm at an intuitive level, and continues with more specific details about the pseudocode and implementation.

High-Level Description

Our algorithm uses a greedy strategy to attempt to minimize the conflicts, making all its decisions in a locally optimal manner. The algorithm bears similarities with FF's extraction algorithm. As such, for a state s in a problem P, it receives as input a Graphplan-style planning graph for P^+. Then, until an RP is found, it will iteratively choose a goal g, then an achiever for g, and will regress the current set of goals through the achiever.

As opposed to FF, when determining which goal to satisfy, our method picks one that can be achieved by an action that introduces the minimum possible conflicts. If the best possible achiever still introduces conflicts, it attempts to fix this situation by performing two subsequent steps: *conflict avoidance* and *conflict repair*.

Conflict avoidance (lines 12-13 in Algorithm 1) attempts to replace the action b added to the relaxed plan at the previous iteration (i.e., the one at the front of the RP suffix before adding a new action at the current iteration). The replacement, if successful, guarantees that the achiever chosen to be added to the RP will introduce fewer conflicts than before the replacement.

Conflict repair (lines 14-17) attempts to reduce the number of conflicts by inserting a "repairing" sequence of actions in the RP. The justification for this step is that it turns out that some conflicts *can* be fixed by adding actions to an RP. To see this, consider the RP suffix of Example 2, and assume that action $drop(c1, a2, a4)$ – which produces add-prevail conflicts – is considered to be prepended to the RP suffix. Here, one can insert a 1-action repair to the head of the current RP suffix, namely $go\text{-}out(hoist)$, such that $drop(c1, a2, a4)$ introduces no conflicts at all when prepended to the resulting suffix.

The conflict repair phase searches for a k-step action sequence that will reduce the conflicts introduced by the currently considered achiever to the *minimum*. In case there are two sequences that reduce the conflicts by the same amount, the one that has cheapest preconditions is chosen, where the cost of the precondition is defined as in FF. Repair is obviously an exponential task, but the search we perform here is quite constrained, considering only conflict-free actions for the repair sequence, and usually employing low values of k.

Algorithm 1 Low-Conflict Relaxed Plan Extraction

```
1: function ExtractPlan(Relaxed planning graph PG, Set of
   goal facts finalGoals, Integer k)
2:   goals ← finalGoals
3:   satGoals ← goals in goals that appear at first level of PG
4:   relPlan ← ()
5:   for level ← depth of PG downto 1 do
6:     thisLevelGoals ← goals
7:     while thisLevelGoals \ satGoals ≠ ∅ do
8:       minConfGoals ← MinConflGoals(thisLevelGoals, level)
9:       g ← Most constrained goal in minConfGoals
10:      a ← min-conflict achiever for g
11:      if a introduces conflicts then          ▷ conflict avoidance
12:        Try to replace relPlan's first action to reduce a's confl.
13:        if replacement succeeded then goto line 8
14:      if a still introduces conflicts then    ▷ conflict repair
15:        M ← BestKStepRepair(k, a, PG, goals, level)
16:      else
17:        M ← ()
18:      relPlan ← a · M · relPlan
19:      goals ← regressed goals for new relPlan
20:      Delete from thisLevelGoals facts achieved by a · M
21:  return relPlan
```

Specific Details

We now proceed to give a more detailed explanation of some aspects of the pseudocode and its implementation. As noted above, the algorithm is inspired by FF's extraction method. As such, it takes as input a planning graph *PG*, a set of goals, and an integer k corresponding to the maximum length of a repair sequence. The algorithm iterates from the last (deepest) level of the graph. In each iteration it achieves all subgoals in *thisLevelGoals*. Note that this variable is initialized with the initial set of goals (line 2). As a consequence, all goals not satisfied at the first level of the graph will be considered for achievement in the first iteration. On subsequent iterations, on each level ℓ we satisfy all subgoals generated by actions at level $\ell + 1$. This is a fundamental difference with FF, since FF's algorithm achieves at level ℓ only those goals that first appear at level ℓ. The justification for our modification is simple: achieving a goal at a higher level in the graph provides a broader spectrum of achievers to choose from, and thus decreases the chances of finding only high-conflict achievers.

To select the next goal to achieve, g, goals that have achievers that would introduce the fewest conflicts are computed in line 8 and stored in *minConfGoals*. The most constrained goals are those in *minConfGoals* that have the fewest achievers. In line 9, g is set to a most constrained goal, with ties broken by choosing the goal that appears at the highest level of *PG* for the first time. Remaining ties are broken arbitrarily. The best achiever for g, i.e. one that produces the fewest conflicts, is then chosen (line 10). In case of ties, achievers with lower precondition costs are preferred.

The implementation of the replacement of an action b in the conflict avoidance phase (line 12) deserves additional comments. To implement it, we keep track of the set of min-conflict actions that have been considered at the time when b has been selected. If an action is found that reduces

the conflicts introduced by the current best achiever, such an action will replace b in *relPlan*. If the replacement is successful (line 13), the control jumps to line 8. This allows the algorithm to choose a *best available* new achiever by reconsidering new potential min-conflict actions.

For conflict repair, the BESTKSTEPREPAIR function uses regression search for a sequence of non-conflicting actions that *achieves* those facts in *thisLevelGoals* that are in conflict with a. Any action may be considered for this search (not just those in the *PG* of the current state). The sequence sought, besides strictly reducing the conflicts produced by a, must be such that its preconditions either are added by a or occur in *PG* at a depth lower than *level*.

A very important note has to do with the input relaxed planning graph PG. While developing the algorithm we noticed that sometimes actions that produce fewer conflicts do not appear in a planning graph that is expanded just until all goals are reached. To attempt to minimize conflicts even more, our planner may consider graphs that are expanded beyond the goal's level.

The planner determines how many extra levels to expanded before starting to plan, and commits to this value throughout the search for a plan. To this end, the current implementation uses a very simple strategy. For the initial search state only, it runs Algorithm 1 for planning graphs expanded to increasing levels beyond the goals, stopping when a fixed point in the number of conflicts in the RP is found. The extra value up to which to expand the graph is set to the level at which the fixed point is reached.

Proposition 2 *Algorithm 1 is polynomial in the size of the problem P and exponential in the parameter k.*

Although the algorithm is exponential in k, we see in the experimental evaluation that good results can be obtained by setting k to values as low as 1 or even 0.

Lookahead Macros Revisited

An RP can sometimes be too optimistic and therefore much shorter than a real plan. When building a lookahead macro, YAHSP's plan repair strategy – which we call *repair by substitution – replaces* an existing action with one or more actions. The number of replacement actions is bounded by the number of add effects of the replaced action.

When RPs are computed in a regression fashion, as is the case in FF and in our new algorithm, it can often happen that the tail of a relaxed plan is conflict-free. As more actions are prepended and an RP becomes more complex, conflicts are harder to avoid towards the beginning of a relaxed plan. Conflicts that occur early in an RP can limit the length of the lookahead macros extracted from that RP.

To obtain longer lookahead macros, we introduce a simple, yet effective extension called *repair by insertion*. When generating the lookahead macro from a state s, if no action from the remaining part of an RP can be appended to the current macro prefix m, a new action a is sought. The conditions that an action a needs to satisfy are that it can be added to m (i.e., it is executable in $\gamma(s, m)$, the final state of m), and that its effects allow to further append to the macro at least one action from the remaining part of the RP. The repair by insertion strategy is shown in Algorithm 2. Other details of the algorithm for building lookahead macros are similar to the original YAHSP procedure, which we outline in the background section. For more details, see the original paper (Vidal 2004).

Algorithm 2 Repair by insertion.

```
1: function SEEKREPAIRACTION(m, s', R)
2:   C ← ∅                          ▷ set of candidate repair actions
3:   for each i ∈ [1 ... |R|] do
4:     C ← C ∪ SEEKCANDIDATES(m, s', R_i)
5:   return arg max_{a∈C} score(a)
```

The arguments to SEEKREPAIRACTION are m, the macro prefix built so far, $s' = \gamma(s, m)$ and R, the remaining part of $RP(s)$. R_i is the suffix of R starting from the i-th position, and $first(R)$ denotes the first action in R. SEEKCANDIDATES returns all actions a that satisfy all following conditions: (1) $a \in \mathcal{A}(s')$; (2) $first(R_i) \in \mathcal{A}(\gamma(s', a))$; and (3) a does not introduce a cycle (i.e., a state repetition) in m. Each action returned by SEEKCANDIDATES is assigned a score defined as $|add(a) \cap pre(R_i)| - |del(a) \cap pre(R_i)|$. The expression $pre(R_i)$ corresponds to the precondition of the sequence of actions R_i, which is formally equivalent to $Regr(R_i, \emptyset)$. We introduce a new heuristic criterion (score) to select a winner among all candidates for insertion, preferring an action that adds more and removes fewer global preconditions of R_i. This rule is used to increase the chances that more steps from R_i – not only the first one – could be appended to m.

Experimental Evaluation

We implemented our RP extraction algorithm on top of FF-v2.3. We also implemented 2 lookahead strategies: (1) the lookahead with insertion repairs (*LA-ins*), as presented in the previous section and (2) the YAHSP lookahead (*LA-yahsp*), as described in the original paper. In the remainder of the section, we use h^{cm} to refer to our heuristic, and h^{FF} to refer to the FF heuristic. Note that each time we use h^{FF} we actually are using Hoffmann and Nebel's implementation of h^{FF}, since our techniques are implemented on top of FF.

For the evaluation, we use FF's search algorithm: enforced hill-climbing algorithm (EHC) followed by a best-first search (BFS) in case EHC fails. We extend EHC and BFS with lookahead macros such that each search node has an additional successor that is obtained by applying the lookahead macro. As in YAHSP, the helpful actions considered when using our RPs correspond to the set of actions in the RP that are applicable in the current state. Obviously, the macro-successor is not pruned in helpful action filtering.

Using 19 benchmarks for classical planning,[3] we compare the performance of h^{FF} and h^{cm} using three lookahead configurations (plain EHC/BFS, EHC/BFS plus *LA-yahsp*, and EHC/BFS plus *LA-ins*), with 4 values for k $(0, \ldots, 3)$. Our

[3]We have used benchmark problems available online. We removed action costs from the woodworking domain (IPC-2008). For the matching-bw domain (IPC-2008, learning track) we used 30 bootstrap problems plus 30 (out of the 60) target problems.

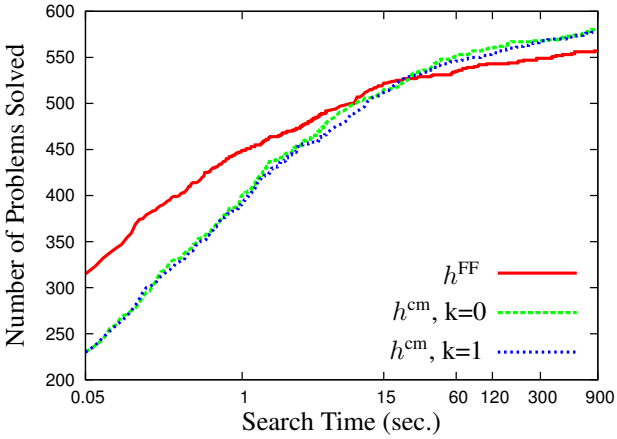

Figure 2: Number of planning instances solved by the three different RP techniques, using *LA-ins* as lookahead.

best results – in terms of number of problems solved – were obtained when $k = 0$ or $k = 1$, and therefore we only report on those here. For higher values, we obtained good results in some domains but the high overhead incurred in the computation of h^{cm} did not pay off in the long run. As a conclusion, for our input data, a simple choice such as not performing k-step repairs is a good strategy. However, we don't conclude that k-step repairs would be useless in general.

All experiments were run on 2.8 GHz AMD Opteron cores, running Linux. The limits of 1.5 GB of RAM, and 900 seconds of time were imposed on each instance.

The experiments reported below focus on a few main directions. First, we analyze the performance of our best system version, which includes both enhancements. Then we break down the analysis to assess the contribution of each individual enhancement.

Best Configuration Table 1 summarizes the results of our best configuration: h^{cm} with *LA-ins* using $k = 0$. For comparison, we include data for h^{FF} (also with *LA-ins*), and the percentage of problems solved by the planning systems LAMA and YAHSP. Results for these planners are included as a reference to the state of the art. Since the search algorithms used by them are different, it is not possible to do a good comparison of their heuristics and ours. Furthermore, the LAMA planner uses other techniques (i.e., landmark computation) in addition to also exploiting the FF heuristic.

We observe that in 6 out of 19 domains our approach solves more problems than h^{FF} using the same lookahead strategy, whereas h^{FF} is better in 3 domains. Moreover, in 16 out of 19 domains, h^{cm} is equally or more informed than h^{FF}. The informedness is measured here using the geometric mean of the ratio between the states expanded by each of h^{FF} and h^{cm} using *LA-ins*. The combination h^{cm} and *LA-ins* has the best overall success percentage (89.88%), outperforming YAHSP, LAMA and the system version that uses h^{FF}.

In terms of CPU time, h^{cm} is slower to compute than h^{FF}. A rough average value, computed over all domains and all instances, indicates is h^{cm} is 4 times slower per node evaluated. Despite this, h^{cm} has a better overall performance, since the savings in the expanded nodes are often greater

Domain	s-ratio **mean**	ℓ-ratio **mean**	h^{FF} **% sol**	h^{cm} **% sol**	YAHSP **% sol**	LAMA **%sol**
woodworking	6.07	1.61	93	100	62	100
storage	3.81	1.34	60	100	67	60
matching-bw	2.95	1.09	62	67	*	93
pipesw-notan	2.36	1	86	94	100	88
pipesw-tan	1.61	1	60	80	42	80
satellite	1.4	0.99	100	97	100	83
depots	1.4	0.97	100	100	82	95
mystery-adl	1.36	0.96	50	47	60	63
driverlog	1.26	0.91	95	100	100	100
freecell	1.24	0.91	98	90	95	97
TPP	1.15	0.89	100	100	100	100
blocks	1.11	0.88	100	100	100	100
zenotravel	1.06	0.87	100	100	100	100
mprime-strips	1.05	0.87	100	100	100	100
philosophers	1	0.85	100	100	*	56
gripper	1	0.81	100	100	100	100
logistics	0.97	0.74	96	96	96	96
rovers	0.81	0.73	100	100	100	100
trucks	0.19	0.56	37	37	*	50
Average	1.33	0.92	86.17	**89.88**	87.75	87.52

Table 1: Geometric mean of the ratio between states expanded (s-ratio) and plan length (ℓ-ratio) obtained by h^{FF} and h^{cm} using *LA-ins*, followed by the percentage of problems in which a plan is found (% sol), for *LA-ins* with h^{FF}, *LA-ins* with h^{cm}, YAHSP and LAMA. YAHSP was run in "lobfs" mode. Above the line are those domains in which h^{cm} is more informed than h^{FF} (i.e., s-ratio ≥ 1). ℓ-ratio ≥ 1 iff h^{cm} finds shorter or equal solutions on average. "*" means the planner does not handle that domain. YAHSP average values are on a different set of domains.

than the time overhead per node. Figure 2 shows the number of problems solved as the search time (per instance) increases. We observe that, due to the higher overhead incurred by h^{cm}, h^{FF} is better during the first few seconds of searching. Soon after 15 seconds, h^{cm} catches up, solving more problems. The time range where h^{FF} solves more problems than h^{cm} is short (about 15 out of 900 seconds), despite the rather opposite visual effect created by using a logarithmic horizontal scale. We use a logarithmic scale for a better readability. With a normal scale, the [0,15] interval would be almost invisible.

Finally, in terms of solution quality (length), in problems that can be solved by both heuristics, h^{cm} tends to produce longer plans – 8% longer in average. In 10 out of the 19 domains, half of the solutions obtained by h^{cm} are shorter than those obtained by h^{FF}.

The best value for the k parameter is 0. Using $k = 1$ yields comparable, but slightly worse results (Figure 2). The average success ratio drops slightly from 89.88% to 89.31%.

We switch our attention to evaluating how various system features contribute to the overall performance. With lookahead turned on, h^{cm} provides both a heuristic value for the state at hand, and a set of helpful actions (including the lookahead macro). To evaluate the impact of each of these two ways of employing h^{cm}, an "intermediate" system configuration was run. We use the standard h^{FF} to provide heuristic state estimations, and h^{cm} in combination with *LA-ins* to compute lookahead macros and helpful actions. De-

spite the overhead of evaluating states twice, the resulting configuration works well. It has an average success rate of 88.56%, which places it right in between the two configurations (h^{FF} and h^{cm}) reported in Table 1. The result suggests that both ways of using h^{cm} have a positive contribution to the system performance, a conclusion that is supported by the following experiments too.

h^{cm} vs. h^{FF} with No Lookahead We continue our analysis by evaluating each of the two enhancements (h^{cm} and *LA-ins*) individually. First, we compare the performance of h^{cm} and h^{FF} with the lookahead switched off. The best results were obtained for h^{cm} with $k = 1$. A summary of the results is shown in Table 2.

With no lookahead, h^{cm} is competitive with and even slightly better than h^{FF}. This time, however, the overall differences between h^{cm} and h^{FF} are smaller than in the case where lookahead is switched on, which was analyzed earlier (Table 1). This behaviour seems to indicate that h^{cm} offers a better pool of actions to be exploited in a lookahead macro.

Finally, in terms of solution quality (length), in problems that can be solved by both heuristics, h^{cm} tends to produce longer plans. On average, there is a 10% increase. In 9 out of the 19 domains at least half of the solutions obtained by h^{cm} are shorter than those obtained by h^{FF}.

LA-ins* vs. *LA-yahsp To evaluate the impact of the *LA-ins* lookahead, we compare *LA-ins* and *LA-yahsp* using the standard heuristic h^{FF}. *LA-ins* outperforms *LA-yahsp*. Indeed, in 9 of 19 domains *LA-ins* solves more problems than *LA-yahsp*, and in no domain it solves fewer instances. Moreover, in 16 domains, *LA-ins* expands the same or fewer nodes than *LA-yahsp* on average. Even though *LA-ins* yields a search that expands fewer nodes, there is no compromise in solution quality. Solutions produced by *LA-ins* are only 1% longer on average. It is important to note, however, that solutions obtained with lookahead are generally of a lower quality than those obtained without lookahead (Vidal 2004). Compared to plain EHC/BFS (i.e., the FF planner), *LA-ins* obtains solutions that are 20% longer on average. In terms of time, as both lookahead strategies incur a very similar overhead, search times very much correlate to the number of evaluated nodes, and thus are better for *LA-ins*. In terms of problems solved, *LA-ins* plus EHC/BFS outperforms the FF planner, solving more instances in 10 out of the 19 domains, and solving fewer instances in only one domain.

We note also that our implementation of the YAHSP lookahead (*LA-yahsp*) used with h^{FF} is competitive with the YAHSP planner. Considering only domains that can be handled by both planners, *LA-yahsp* has a success ratio of 88% compared to the 87.75% of YAHSP. *LA-yahsp* produces slightly longer plans; on average, 2% longer. These differences in the results are explained by the fact that (1) YAHSP uses a slight modification of the FF relaxed plan extraction, and (2) YAHSP uses a different search algorithm.

Finally, we briefly remark that *LA-yahsp* performs well in combination with h^{cm}, but not better than *LA-ins*. Its success rate, 87.47%, is lower than that of *LA-ins* (Table 1). Note that *LA-yahsp* can exploit h^{cm} better than h^{FF}. In fact, *LA-yahsp* with h^{FF} has an average success rate of 83.77%.

	FF	h^{cm}	State Ratio		Length Ratio	
Domain	**% sol**	**% sol**	**mean**	**median**	**mean**	**median**
storage	57	**67**	3.84	1.71	0.99	1
pipesw-notan	72	**82**	3.62	3.44	1	1
matching-bw	62	**63**	3.6	3.46	0.97	1
pipesw-tan	40	**52**	2.55	2.11	0.89	0.86
rovers	**100**	**100**	2.22	2.05	0.98	1
mprime-strips	**97**	90	1.46	1.33	0.93	0.93
depots	**100**	91	1.41	0.58	0.86	0.84
satellite	**100**	97	1.31	1.19	0.95	0.99
woodworking	59	**100**	1.24	1	1.01	1
blocks	89	**91**	1.09	1	0.74	0.77
TPP	**87**	80	1.02	0.95	1	1
driverlog	80	**85**	0.65	0.68	0.95	1
mystery-adl	**53**	50	0.62	0.8	0.93	1
freecell	**98**	75	0.6	0.68	0.77	0.79
philosophers	**25**	21	0.57	0.56	1	1
zenotravel	**100**	**100**	0.41	0.48	0.8	0.85
logistics	**96**	**96**	0.35	0.4	0.8	0.81
gripper	**100**	**100**	0.28	0.26	0.77	0.76
trucks	37	**43**	0.19	0.26	0.9	0.9
Average	76.35	**78.11**	1.01	0.92	0.9	0.92

Table 2: Percentage of problems solved (% sol), and the geometric mean and median of the ratio between states expanded and the ratio between plan lengths obtained by h^{FF} and h^{cm}. As in Table 1, state/length ratios over 1 indicate better performance for h^{cm}. Above the line, those domains in which, on average, h^{cm} is more informed than h^{FF}.

Experimental Conclusions In summary, our experimental results show that, under several search settings, h^{cm} outperforms h^{FF} in terms of success rate. The benefits of using h^{cm} come from two sources: first, from the exploitation of the contents in the low-conflict RPs provided by h^{cm}, and second, from the sometimes more informed heuristic value provided by h^{cm}.

h^{cm} produces best results in combination with our *LA-ins* lookahead. The results, in terms of success ratios, are state-of-the-art, with significantly improved performance in a few domains in which planners like LAMA and/or YAHSP perform poorly (e.g., storage). In terms of solution quality, h^{cm} usually yields slightly longer plans.

Finally, we showed that *LA-ins* tends to outperform *LA-yahsp*, regardless of the heuristic used. *LA-ins* and *LA-yahsp* used with h^{FF} are competitive with the YAHSP planner.

The improvements produced by h^{cm} are not consistent across all domains. We do not have conclusive insights as to why this occurs. Clearly, key aspects of some domains are captured by conflict minimization; e.g., in storage this leads to additional helpful actions. In other domains (e.g., freecell) it seems that the *order* in which goals are achieved for the RP construction is not adequate, yielding a poor RP.

Related Work

Other researchers have also focused on addressing some of the shortcomings of the FF relaxed plan heuristic. Nguyen and Kambhampati (2000) propose several heuristics that combine various measures obtained from a planning graph (with mutex information) built for the unrelaxed problem, thus taking some conflicts into account. The heuristic in the

SAPA temporal planner uses similar techniques to obtain an improved estimate of plan makespan (Do and Kambhampati 2003). Yoon, Fern, and Givan (2006) use machine learning techniques to learn (approximate) mappings from features of the problem and relaxed plan to the true plan length, given a number of solutions to problem instances belonging to the same class. In contrast to our work, these efforts focus only on improving the heuristic distance estimate, rather than improving the correctness of the relaxed plan itself.

Hoffmann and Geffner (2003) use flaw removal as a criterion to guide the main search in a Graphplan framework. In contrast, we aim at flaw minimization in a greedy, cheap procedure that is used to compute relaxed plans.

Works by Benton, van den Briel, and Kambhampati (2007), and by Coles et al. (2008) modify the standard FF relaxed plan extraction to produce relaxed plans that more closely correspond with linear programming (LP) relaxations of the original problem. Their methods however, do not exploit conflicts as we do, and are tailored for different planning applications: the former deals with partial satisfaction planning, and the latter with numeric planning. In classical planning applications, Coles and Smith (2006) exploit knowledge about *generic object types* to recognize when actions need to be added to the relaxed plan, thus improving its fidelity. While this approach is domain-independent, as type information is automatically extracted from the problem description, it can only improve relaxed plans in problems where objects with the prespecified generic types are present.

Conclusion and Future Work

In this paper we have proposed the utilization of low-conflict RPs as an enhancement of classical planning. A low-conflict plan is an approximate solution to an optimization problem that minimizes the conflicts in an RP. We proposed an algorithm that is able to quickly obtain reasonable low-conflict RPs. The heuristics obtained using our RPs can improve sometimes significantly over the standard FF RP heuristic. We showed that state-of-the-art performance is obtained when using lookahead policies for the construction of successors. The resulting heuristic provides notably good performance in selected domains (e.g., storage), in which it seems to capture aspects of the problem that are oversimplified by other heuristic-search approaches. In addition, we have proposed a procedure to construct lookahead policies, that consistently outperforms previous approaches.

Low-conflict RPs do not seem to provide the same benefits across all domains. Our RPs seem to be particularly useful in applications in which ignoring conflicts imply ignoring critical information about the solution to the problem (like, for example, omitting a useful action from the set of helpful actions). An investigation of the key aspects of a domain that can be exploited by these RPs is an intriguing problem, but is out of the scope of this paper.

We view our algorithm as one step ahead but, obviously, not the final answer in addressing the optimization problem defined in this paper. In the future, we could investigate the practicality of using other approaches (e.g., LP) to approximate the optimization problem. Also, there are other applications in which low-conflict RPs could be useful. For example, they could be utilized with satisfiability planners, which are usually able to compute makespan-optimal plans.

Acknowledgements We thank Fahiem Bacchus, Christian Fritz, Patrik Haslum, and Sheila McIlraith for valuable discussions and comments on drafts of this paper. We also thank the reviewers for their very useful comments. J. Baier acknowledges funding from NSERC (Natural Sciences and Eng. Research Council of Canada) and ERA (Ontario Early Research Award). NICTA is funded through the Australian government's *backing Australia's ability* initiative.

References

Benton, J.; van den Briel, M.; and Kambhampati, S. 2007. A hybrid linear programming and relaxed plan heuristic for partial satisfaction problems. In *ICAPS*, 34–41.

Blum, A., and Furst, M. L. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* 90(1-2):281–300.

Botea, A.; Müller, M.; and Schaeffer, J. 2007. Fast Planning with Iterative Macros. In *IJCAI*, 1828–1833.

Coles, A., and Smith, A. 2006. Generic Types and their Use in Improving the Quality of Search Heuristics. In *PlanSIG*.

Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. A Hybrid Relaxed Planning Graph-LP Heuristic for Numeric Planning Domains. In *ICAPS*, 52–59.

Do, M., and Kambhampati, S. 2003. Sapa: A Scalable Multi-Objective Metric Temporal Planner. *Journal of Artificial Intelligence Research* 20:155–194.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *AIPS*, 140–149.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *ICAPS*, 140–147.

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

Hoffmann, J., and Geffner, H. 2003. Branching Matters: Alternative Branching in Graphplan. In *ICAPS*, 22–31.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.

Hsu, C.-W.; Wah, B.; Huang, R.; and Chen, Y. 2007. Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *IJCAI*, 1924–1929.

McAllester, D. A., and Rosenblitt, D. 1991. Systematic Nonlinear Planning. In *AAAI*, volume 2, 634–639.

Nguyen, X., and Kambhampati, S. 2000. Extracting Effective and Admissible State Space Heuristics from the Planning Graph. In *AAAI*, 798–805.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, 975–982.

Vidal, V. 2004. A Lookahead Strategy for Heuristic Search Planning. In *ICAPS*, 150–159.

Yoon, S.; Fern, A.; and Givan, R. 2006. Learning Heuristic Functions from Relaxed Plans. In *ICAPS*, 162–170.