

Plan Repair as an Extension of Planning

Roman van der Krogt* and Mathijs de Weerdt

Delft University of Technology

The Netherlands

{r.p.j.vanderkrogt|m.m.deweerd}@ewi.tudelft.nl

Abstract

In dynamic environments, agents have to deal with changing situations. In these cases, repairing a plan is often more efficient than planning from scratch, but existing planning techniques are more advanced than existing plan repair techniques. Therefore, we propose a straightforward method to extend planning techniques such that they are able to repair plans. This is possible, because plan repair consists of two different operations: (i) removing obstructing constraints (such as actions) from the plan, and (ii) adding actions to achieve the goals. Adding actions is similar to planning, but as we demonstrate, planning heuristics can also be used for removing constraints, which we call unrefinement. We present a plan repair template that reflects these two operations, and we present a heuristic for unrefinement that can make use of an arbitrary existing planning technique. We apply this method to an existing planning system (VHPOP) resulting in POPR, a plan repair system that performs much better than replanning from scratch, and also significantly better than another recent plan repair method (GPG). Furthermore, we show that the plan repair template is a generalisation of existing plan repair methods.

Introduction

When plan construction is completed, our work begins. Any agent executing a plan should be monitoring its environment and the actual effects of its actions, because there are more ways in which a plan can go wrong than there are actions an agent can execute. In many occasions, the agent needs to alter parts of its plan to be able to still attain its goals. We know that in theory modifying an existing plan is no more efficient than a complete replanning (from the current state) in the worst case (Nebel & Koehler 1995). However, we expect that in practice plan repair is often more efficient, since a large part of the plan usually is still valid. On top of this, in many problem domains it may be quite costly to change your whole plan, for example because of bookings or commitments to other agents that have been made based on the original plan. Furthermore, in mixed initiative settings, the user might more easily accept a plan that was repaired (and thus resembles the original plan) over a plan that was created from scratch (and might look entirely different). These

*Supported by the Freight Transport Automation and Multi-Modality (FTAM) program of the TRAIL research school for Transport, Infrastructure and Logistics.

considerations directly lead to the following problem: how can an existing plan be repaired such that the (updated) set of goals can be obtained from the (updated) initial state?

Currently, already quite a number of systems that do some form of plan repair exists. However, there are many more planning systems than there are ways to deal with plan repair. Moreover, in general, these planning systems perform much better than the average plan repair system. Plan repair technology could benefit a lot from the knowledge and bright ideas in the AI planning research community, if there was a straightforward way to use these in plan repair systems. Such a method to do plan repair using an extension of (normal) planning technology is exactly the main contribution of this paper.

In our approach a couple of ideas come together. First, because of bookings and commitments to others, plans that are not too different from the original plan are preferable. Therefore, the plan repair process should start with the original plan. Second, two different operations in plan repair can be distinguished: (i) removing actions from the plan that actually make it harder to attain the goal(s), and (ii) adding actions that bring the agent closer to the goal. Observe that the latter operation is similar to planning. Our third and final idea is that planning also can be used for the former operation: heuristics similar to planning can be used to determine whether an action should be removed.

In the next two sections these ideas are explained in more detail. First, a general template is given that describes the dissection into adding and removing actions, and, next, a method is presented for removing exactly those actions from the plan that are expected to have a negative influence on the plan length. In the fourth section we discuss how this algorithm can be combined with the VHPOP planner (Younes & Simmons 2003) using the general template. The performance of the resulting plan repair system, which we call POPR, is compared to (re)planning from scratch and to another recent plan repair method, GPG (Gerevini & Serina 2000). Finally, we show that other plan repair systems can be seen as specific instances of our general plan repair template. First, however, we give a short summary of refinement planning.

A Template for Plan Repair

Refinement Planning

The construction of a plan can be seen as an iterative refinement of the set of all possible plans. This view is called *refinement planning* (Kambhampati, Knoblock, & Yang 1995; Kambhampati 1997). It is shown that most existing (classical) planning algorithms can be conceived in this way. The idea behind refinement planning is that we start with the set of *all* possible sequences of actions and reduce this set by adding constraints (such as “all plans in this set should at least have this specific action”). Besides actions, such constraints can impose an ordering or specify that a particular proposition should hold at a specific point (called *point truth constraints*, or PTC, by Kambhampati) or during a specific interval (*interval preservation constraints*, IPC). We keep adding further constraints until all plans that match the constraints are solutions to the planning problem. During this refinement, not this set of all *candidate* plans is stored, but the constraints are stored in a so-called *partial plan*. Given a partial plan P , the set of candidates it represents is denoted by $candidates(P)$.

A *refinement strategy* defines how a partial plan is to be extended and the set of candidates thus refined. A refinement strategy \mathcal{R} is a function that maps a partial plan P to a set of partial plans $\mathcal{P} = \{P_1, \dots, P_n\}$, such that for each of the new partial plans, the candidate set is a subset of $candidates(P)$. A template for a general refinement planner looks as follows: starting with an empty constraint set, represented by an empty partial plan, say P , check whether a minimal candidate of P is a solution to the problem at hand. If so, we are done. If not, we apply a refinement strategy \mathcal{R} to obtain a collection of partial plans $\mathcal{P} = \mathcal{R}(P)$ where each partial plan has a different additional constraint with respect to P . Select a component $P' \in \mathcal{P}$ and check again whether a minimal candidate of this partial plan is a solution and apply \mathcal{R} again if not. Proceed until a solution is obtained, or the set of partial plans is empty, in which case we backtrack.

Unrefinement Planning

For plan repair, we cannot use the same template directly, because a refinement strategy can only *add* constraints. If this restriction is relaxed, refinement planning can be seen as a unifying view on both planning and plan repair (van der Krogt, de Weerd, & Witteveen 2003). However, this is not very elegant, and, more importantly, it hides the fact that plan repair really constitutes two separate activities: removing actions and other constraints from the plan that are obstructing the successful alteration of the plan, and the (often subsequent) expanding of the plan to include actions solving the planning problem. Therefore, we propose to include an *unrefinement strategy* for plan repair in the template algorithm.

The main idea behind our plan repair template is that plan repair consists of two phases (that can occur in any permutation, depending on the particular method): the first phase involves the removal of constraints from the current partial plan that inhibit the plan from reaching its goals. The second phase is a regular planning phase, in which the partial

Algorithm 1 PLAN REPAIR (P, Π, \mathcal{H})

Input: A partial plan P , a problem Π and a history \mathcal{H}

Output: A solution to Π or ‘fail’

begin

1. **if** $candidates(P)$ is empty **then**
 - 1.1. **return** fail.
2. **if** $solution(P, \Pi)$ returns a solution Δ **then**
 - 2.1. **return** Δ .
3. **if** we choose to unrefine **then**
 - 3.1. Select unrefinement strategy \mathcal{D} and generate new plan set $\langle \mathcal{P}, \mathcal{H}' \rangle = \mathcal{D}(P, \mathcal{H})$.
4. **else**
 - 4.1. Select refinement strategy \mathcal{R} and generate new plan set $\langle \mathcal{P}, \mathcal{H}' \rangle = \mathcal{R}(P, \mathcal{H})$.
5. Non-deterministically select a component $P_i \in \mathcal{P}$ and call PLAN REPAIR(P_i, Π, \mathcal{H}').

end

plan is extended (refined) to satisfy the goals. For example, suppose that we have a plan for driving to a meeting by car. However, upon walking to the car we see that one of its tyres is flat. A simple repair for this plan could be to add actions that change the tyre with a spare one, and drive to the meeting as planned. Now suppose this is a very important meeting at which you do not wish to come late. In that case, replacing the tyre could cost too much time. Instead, it would be better to remove the drive action from the plan, and to replace it with actions using a taxi for transportation. Thus, to repair a plan, a planner should not only employ a *refinement* strategy for extending the plan with actions that will reach the goals (such as replacing the tyres in the example). Planners should also employ an *unrefinement* strategy for retracting constraints from the partial plan (removing actions from a plan that are obstructing a proper solution, such as the drive actions in the previous example).

An extension of the refinement planning template that allows for unrefinement strategies to be employed, can be found in Algorithm 1. This plan repair template differs from refinement planning in only two ways. First, we choose between *unrefining* the plan, i.e. removing refinements (constraints), or *refining* the plan, i.e. adding refinements. For unrefining a plan we select an unrefinement strategy \mathcal{D} and apply it to the partial plan P (step 3.1). Refinement takes place as in the regular refinement planning approach (step 4.1). Second, we introduce a *history* \mathcal{H} that some approaches require to keep track of the refinements and unrefinements they have made, in order to be able to prevent doing double work (and endless loops). Each call to a refinement or unrefinement strategy updates the history to reflect which partial plans have already been considered. Techniques like Tabu-search (Glover & Laguna 1993) may be employed to best make use of this available memory. In the next section, we present a method to repair plans in which we prevent loops by using a straightforward search, so we do not need to keep track of an explicit history of refinements and unrefinements.

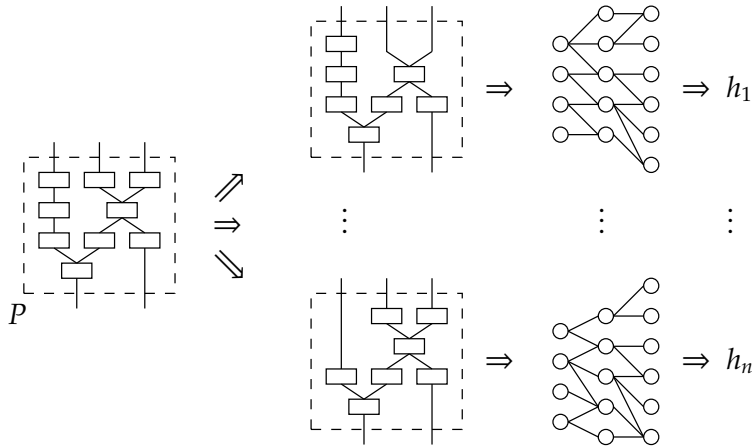


Figure 1: Sketch of the unrefinement heuristic. From the original plan P on the left, we derive n subplans and calculate heuristic values (h_1, \dots, h_n) using (in this case) a planning graph heuristic.

A Plan Repair Algorithm Using Refinement Planners

In this section we present a method that can reuse an existing *planning* heuristic to incorporate plan repair in planners. The planning heuristic that we use in our unrefinement strategy is arbitrary, as long as it can evaluate partial plans for their fitness (i.e. attach a value to a given partial plan indicating how close to a solution it is). In the resulting system, the refinement and unrefinement strategies can coexist without any problems, because the unrefinement heuristic makes use of the refinement heuristic to calculate the heuristic values. This means that using our method, we can add plan repair capabilities to existing planners that use a suitable heuristic. This has the additional benefit that our method can be easily upgraded when more efficient planning heuristics are devised.

Overview

Our approach to using existing planning heuristics for plan repair is sketched in Figure 1. On the left-hand side, we have the current plan P that is to be unrefined. We compute a number of plans that result from *removing* actions from P .¹ For each of these resulting plans, we use the chosen planning heuristic (for example, a planning graph heuristic) to estimate the amount of work required to transform this plan into a valid plan, i.e. for each of these plans a heuristic value is calculated. The plan that has the best (lowest) heuristic value is selected and the (refinement) planning heuristic is used to complete this plan. If the planner cannot produce a solution (which may happen because the heuristic is not perfect), another unrefinement is chosen. Note that we only apply the unrefinement step to the initial plan P . We do *not* unrefine partial plans that have been produced by the refinement strategy, hence this method does not overly enlarge our search space.

¹Note that we consider only removing actions, and not other constraints such as precedence or IPC constraints.

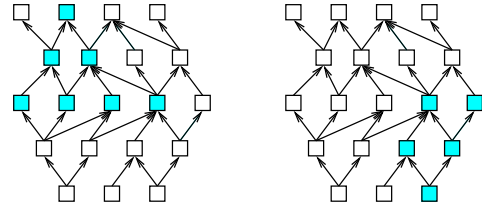


Figure 2: A backward removal tree (left) and a forward tree (right).

Another way to look at this procedure is by considering the search space that is traversed. Initially, the plan repair method is given the current plan P to adapt. It may very well be that this plan is located in a part of the search space in which it is very hard to find a solution by refinement (i.e. only adding actions and constraints), if such a solution exists at all. Our unrefinement heuristic calculates a number of plans by removing actions from P , and uses a planning heuristic to evaluate the conditions of the search space around these partial plans (i.e. it calculates a heuristic value that tells something about the ease with which the plan can be extended). Having identified a better location, we start the refinement process from there. The steps of this procedure are now discussed in greater detail.

Removal Trees

The first step is to decide *which* actions we consider for removal (and thus, for which plans we would like to calculate the heuristic value). Ideally, we would like to consider all possible combinations of actions. However, there is an exponential number of such combinations, so it is clearly too much work to consider all of them. Therefore, we only consider removing certain sets of actions, focusing on actions that are either depending on the initial state, or actions that produce goals or unused positive effects. The idea is that these actions are on the borders of the plan, and that

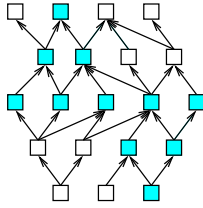


Figure 3: The merger of the trees of Figure 2.

by removing them, we shrink the plan from the outside in. More specifically, we consider a dependency graph of all actions, i.e. we conceive a plan as a directed graph, in which each node is an action, and edges connect actions when the first action produces an effect that satisfies the precondition of the second action (thus, edges represent causal links). From this graph, we extract certain subgraphs, called *removal trees*. A removal tree can either be *forwards* or *backwards*. A forward tree is rooted in an action depending on the initial state; backward removal trees are rooted in an action producing a goal, or of which the effects are not used at all. The *height* of the tree determines which actions are selected in the graph. The following rules determine the actions in a removal tree:

1. For an action o , either depending on the initial state, or producing a goal effect, the removal tree of height 1 consists of the action o itself.
2. For an action o depending on the initial state, the removal tree of height $n + 1$ ($n \geq 1$) of o is defined to be the subgraph generated by the actions in the removal tree of height n , as well as the actions immediately depending on these actions.
3. For actions o producing goal effects, the removal tree of height $n + 1$ ($n \geq 1$) of o equals the subgraph generated by the actions in the removal tree of o of height n , as well as the actions that they immediately depend upon.

Figure 2 shows two examples of removal trees of three levels (shown in grey). The removal tree at the top is rooted in an action producing unused effects, hence this is a downward tree, consisting of actions that the root (indirectly) depends upon. The tree at the bottom is an upward removal tree, containing actions depending upon the root action.

If we only considered the removal trees as unrefinements, however, we would miss out on important unrefinements. For example, we would never consider the whole plan to be removed. Therefore, when we calculate the set of removal trees of depth k , we merge the trees that have an overlap. That is, when we are about to consider removal trees of depth k , we first calculate the set of removal trees, and then merge any removal trees in the set that have an overlap. We do this in such a way, that no two removal trees in the resulting set overlap. Thus, if trees T_1 and T_2 overlap, and so do trees T_2 and T_3 , all three trees are merged into one tree (actually, this is not a tree anymore, but a forest), consisting of the actions in T_1 , T_2 and T_3 . For example, consider Figure 3. This figure shows the result of merging the two trees

Algorithm 2 CALCULATE TREES (P, n)

Input: A plan P with initial action a_0 and goal action a_∞ , and a value n

Output: The set of removal trees of depth n

begin

1. $numTrees = 0$
 2. **for** each action a in P **do**
 - 2.1. **if** there exists a causal link $a_0 \xrightarrow{p} a$ **then**
 - 2.1.1. $numTrees = numTrees + 1$
 - 2.1.2. $base_1 = \{a\}$
 - 2.1.3. **for** $j = 1 \dots n$ **do**

mark all actions in $base_j$ with “ $numTrees$ ”

 $base_{j+1} = \{a' \mid \exists a'' \xrightarrow{p'} a' \wedge a'' \in base_j\}$
 - 2.2. **if** there exists a causal link $a \xrightarrow{p} a_\infty$ **or** no causal link $a \xrightarrow{p'} a'$ exists **then**
 - 2.2.1. $numTrees = numTrees + 1$
 - 2.2.2. $base_1 = \{a\}$
 - 2.2.3. **for** $j = 1 \dots n$ **do**

mark all actions in $base_j$ with “ $numTrees$ ”

 $base_{j+1} = \{a' \mid \exists a'' \xrightarrow{p'} a'' \wedge a'' \in base_j\}$
 3. **for** $j = 1 \dots numTrees$ **do**
 - 3.1. create a set $eq_j = \{j\}$
 4. $Eq = \bigcup_{j=1 \dots numTrees} eq_j$
 5. **for** each action a in P with marks $m_1 \dots m_n, n > 1$ **do**
 - 5.1. let E_{m_j} be the set containing m_j , for $j = 1 \dots n$
 - 5.2. let $E' = \{m \mid m \in E_{m_j}, j = 1 \dots n\}$
 - 5.3. $Eq = Eq \setminus \bigcup_{j=1 \dots n} E_{m_j}$
 - 5.4. $Eq = Eq \cup E'$
 6. **for** each $eq_j \in Eq$ **do**
 - 6.1. let T_j be the subplan generated by the actions marked with the marks contained in eq_j
 7. **return** $\bigcup_{j=1 \dots |Eq|} T_j$
- end**
-

shown in Figure 2.²

Computing Removal Trees

The set of removal trees can be calculated efficiently: there is a polynomial number of removal trees given a certain plan (with respect to the size of that plan), and these can be merged in polynomial time. Algorithm 2 describes how removal trees are calculated. Given a plan P and a value n , it returns a set of merged removal trees of depth n . First, it marks actions in the plan as being part of a removal tree. In step 2.1, it calculates the upward removal trees for each action a that is causally dependent upon the initial state. It does so by first marking the action a itself, followed by the actions that causally depend upon a , the actions depending upon those actions, etc., up to n times. The same procedure (but then in the opposite direction) is followed for down-

²Notice that the two trees depicted in Figure 2 are not the only trees of depth 3 in this plan. In fact, the set of merged removal trees of depth 3 contains just one removal tree: the one equal to the whole plan.

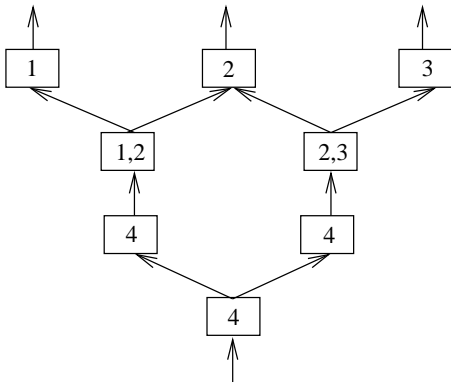


Figure 4: Example plan marked with removal trees labels.

ward trees, rooted in actions that produce goals, or actions of which no effect is used at all. Having marked actions for each removal tree, steps 3 through 5 determine which removal trees are to be merged. The idea is to merge all overlapping removal trees. This is done as follows: first, we calculate a set Eq consisting of singleton sets with the different marks used. Then, in step 5 we successively merge sets with marks that are overlapping. This results in a new set Eq consisting of sets of marks of trees that are to be merged. Finally, in step 6 we compute the removal trees by taking subplans of the plan P generated by the actions.³

Example. Consider the plan in Figure 4, consisting of 8 actions, depicted by boxes (of which the labels can be disregarded for now), and their causal dependencies (the arcs). Three of those actions produce a goal effect, and one of is causally dependent upon the initial state. Therefore, the set of removal trees of depth one consists of 4 trees. We now calculate the set of removal trees of depth 2, using Algorithm 2. The first step is to mark the actions that are part of each removal tree. This results in the markings as shown. A label of “1,2” means that this action is part of both the first and the second removal tree.

To determine which trees are to be merged, we first create the set $Eq = \{\{1\}, \{2\}, \{3\}, \{4\}\}$. Then, we check which actions are marked more than once. Say we first encounter the action marked “1,2”. This results in the sets $\{1\}$ and $\{2\}$ being merged. Now, $Eq = \{\{1,2\}, \{3\}, \{4\}\}$. Secondly (and lastly), we encounter “2,3”, which causes us to merge $\{1,2\}$ (the set containing the label “2”) and $\{3\}$. This results in $Eq = \{\{1,2,3\}, \{4\}\}$. Hence, there are two removal trees of depth 2. One consists of each action marked with either 1, 2 or 3 and the other consists of actions marked with 4.

Selecting a Removal Tree

The set of merged trees is used to calculate possible unrefinements to the plan. Given a (merged) removal tree, the second step is to calculate the heuristic value for that option. To do this, we construct the plan that results when remov-

³Note that for efficiency, steps 3 through 5 can be integrated into steps 2.1 and 2.2.

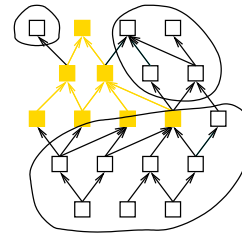


Figure 5: Example cuts of a plan.

ing the removal tree. Next, we can simply apply a planning heuristic to obtain a heuristic value for the plan. However, some heuristics have a problem with calculating a heuristic value for the kind of broken down plans we produce.⁴ To overcome this problem, we automatically construct a special domain on the fly. This domain consists of the original domain, as well as some special actions encoding the plan that we would like to reuse. For this purpose, the plan is broken down into separate parts, called *cuts*. Each cut is chosen such that there are no two actions in a cut that were previously connected through one or more removed actions. For each cut, an action is added which has preconditions and effects equal to the cut. Now, if we calculate a heuristic value for the *empty* plan in this custom domain, the computation includes the “special” actions corresponding to the cuts, effectively producing a heuristic value for the plan from which we constructed the domain. As an example, Figure 5 shows the cuts of the plan that results from removing the removal tree of Figure 2 (left).

The complete algorithm

The complete unrefinement strategy now works as follows: upon detecting that our current plan P is no longer a valid solution to the problem at hand (e.g. when the problem has changed), we begin by computing the removal trees of P of depth one. Overlapping removal trees are merged as discussed before, and for each merged removal tree the resulting plan is computed. For the resulting plans, we consult the planning heuristic to obtain an estimate of the cost of extending the plan to a valid plan. Thus, the planning heuristic is used to select the most promising candidate (if such a candidate exists). This candidate is then passed to the refinement strategy in order to be completed. If this is not possible, the other candidates are tried, until all candidates of this level have been removed. If that happens (or no candidate exists at all), we iteratively increment the depth of the removal trees and try again. This procedure is repeated, until, finally, the whole plan is discarded and a complete replanning is performed.

⁴For example, forward heuristics (such as used in e.g. FF (Hoffmann & Nebel 2001)) expect that the partial plan forms the head of the final plan to be calculated. This means that they assume that actions have to be added *after all* existing actions. In other words, it is not allowed to have actions in the partial plan which do not have their preconditions satisfied. Our heuristic regularly produces partial plans in which this is the case.

Algorithm 3 REPAIR(P, Π, \mathcal{H})*Input:* A partial plan P , a problem Π and a history \mathcal{H} *Output:* A solution to Π or ‘fail’**begin**

1. **if** $\text{candidates}(P)$ is empty **then**
 - 1.1. **return** fail.
2. **if** $\text{solution}(P, \Pi)$ returns a solution Δ **then**
 - 2.1. **return** Δ .
3. **if** we $\mathcal{H} \geq 0$ **then**
 - 3.1. $T = \text{CALCULATE TREES}(P, \mathcal{H})$
 - 3.2. $\mathcal{P}' = \{P - T_i \mid T_i \in T\}$
4. **else**
 - 4.1. generate new plan set $\mathcal{P}' = \mathcal{R}(P)$.
5. Use the planning heuristic to select the components $P_i \in \mathcal{P}$ in a particular order and call REPAIR($P_i, \Pi, -1$).
6. **if** $0 \leq H < |P|$ **then**
 - 6.1. REPAIR($P_i, \Pi, \mathcal{H} + 1$).

end

In Algorithm 3 the procedure described above is cast in the template algorithm as shown in Algorithm 1. Here, we use the history \mathcal{H} to distinguish between the refinement phase and the unrefinement phase of the search. Like the template algorithm, there are three parameters: a plan P to refine or unrefine, the current problem Π and the history \mathcal{H} , which we assume to be 0 when this function is initially called. The first two steps are identical to the steps of the template. In line 3, we decide to refine or unrefine based on the value of \mathcal{H} . If $\mathcal{H} \geq 0$, we calculate the removal trees of depth \mathcal{H} and generate the plan set \mathcal{P}' containing all plans that result from removing a removal tree from the plan P . In the case that $\mathcal{H} < 0$, we apply the chosen refinement strategy to generate further refinements. The non-deterministic selection of components of the new plan set \mathcal{P}' is replaced by a more informed selection based on the planning heuristic. Thus, we enter recursion with the most promising components first. Note that in step 5, we always recurse with a negative value for the history. Hence, we will only select the refinement strategy in subsequent iterations of the algorithm. Finally, if we generated a set of unrefinements in this iteration and none of the recursive calls of step 5 produced a solution, we try to derive a solution using larger removal trees in step 6: unless we arrived at the maximum size of the removal trees (equal to the size of P), we call the REPAIR algorithm with the value of \mathcal{H} increased by one.

Experimental Results

For the experimental validation of our technique, we integrated it into the VHPOP planner (Younes & Simmons 2003).⁵ This planner was chosen since it is a clear refinement planner, that sticks close to the original template al-

⁵That is, we used the refinement algorithm of VHPOP as the refinement operator \mathcal{R} of step 4.1 of Algorithm 3, and the VHPOP heuristic to determine the order in which to try the components of \mathcal{P} in step 5.

gorithm. This makes it easier to add our extensions.⁶ Experiments were performed using the benchmark problem set of GPG (Gerevini & Serina 2000). This benchmark set is already a few years old, but is the only one for plan repair problems that the authors are aware of. It consists of over 250 replanning problems from various often used planning domains: *gripper*, *logistics* and *rocket*. The problems can be divided into 7 sets (2 each for the gripper and rocket domains, and 3 for logistics). Each set contains variants on the same test problem, each with a few changes to the initial state or the goals. For example, the gripper domain features a robot equipped with two grippers. It can move through a number of rooms, and has to move balls from their current location to another. Examples of modifications in this domain are: “ball 2 is located in room B instead of in room A”, or “ball 5 should no longer be brought to A, but to C”.

In our experiments, we compared our system (which we call POPR, for Partial Order Plan Repair) with that of GPG (of which a description can be found in the next section), as we used their benchmark set, and with planning from scratch using VHPOP. Figures 6 to 10 show the run times that were obtained using a Pentium-III running at 1000 MHz. The systems were allowed a maximum of 512 Mb of memory, and 200 seconds of CPU time. All graphs use a logarithmic scale for the CPU times. Note, however, that the graphs do not all have the same scale. Three run times are plotted: one for planning from scratch (using VHPOP, labeled *scratch* in the graph), one for POPR, our version of VHPOP using the proposed plan repair method (labeled POPR) and one for the GPG plan adaptation system (labeled GPG). For brevity, the results of the *gripper* domain are all displayed in one graph; instances 1-30 come from one test set, problems 31-60 come from the other. The same holds for the *rocket* domain.

In general, the results of Figures 6 to 10 show that our plan adaptation system is faster than a complete replanning in all but a few cases. This is especially apparent in the gripper domain, where VHPOP cannot find a solution at all within the time and space limits for certain instances (those reported as 200 seconds in the figure). But also in most of the other domains, the difference is quite clear. The reason for this is that the planning problems that are given to VHPOP after the unrefinement phase are usually much smaller than the complete problem. For some problem instances, however, the instances are such that the VHPOP heuristics lead in the wrong direction when refining a plan that it has received from the unrefinement heuristic. For example, problem 28 of Logistics-A requires backtracking a total of 45 times when planning from scratch, compared to 228 times when performing plan repair.

We shall not discuss the specifics of each of the domains, but limit ourselves to a few observations. Firstly, we point at the oscillating behaviour of our system in the “Rocket” domain. This is caused by the specification changes: for example, for some changes, a passenger is merely required

⁶As an indication of how hard (or easy) it is to adapt an existing planner using this template, we observe that the source code of VHPOP consists of 21,080 lines of code, whereas the adapted system consists of 23,757 lines (and only a few existing lines have been changed or removed).

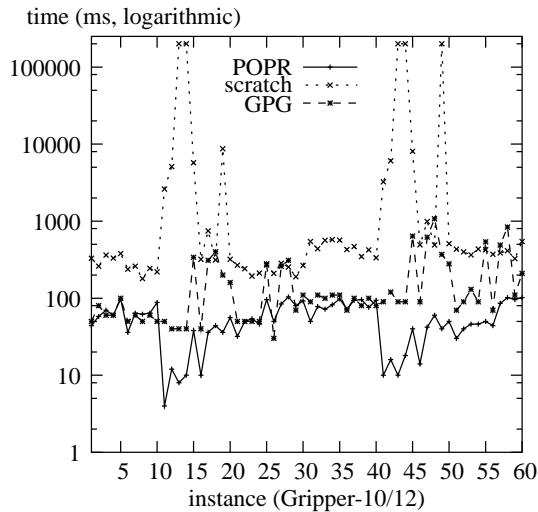


Figure 6: This figure shows the run time in CPU milliseconds (on a logarithmic scale) required by VHPOP from scratch, VHPOP using plan repair, and GPG for the GPG benchmark “Gripper”.

Problem set	POPR	GPG	F	p
Gripper-10/12	55.7	175.8	19.8	< 0.001
Rocket-A/B	132.9	64.3	10.3	< 0.002
Logistics-A	78.1	182.4	50.9	< 0.001
Logistics-B	70	178.8	45.6	< 0.001
Logistics-C	89.5	357.3	9.9	< 0.002

Table 1: Means of POPR and GPG and results of an analysis of variance of the experiments for the different problem sets

to board another rocket (that was flying anyway), whereas for other changes a rocket has to be flown to that person first. In the former case only a small repair is required; in the latter case a larger repair is required. However, while we can attribute the behaviour to the specific instances, we are not entirely sure yet why this has such a big effect in this domain. Surely, the same situation is present in the other domains, but there it does not lead to such big fluctuations.

Another interesting observation can be made in the “Logistics” domains. It can best be seen in Figure 8, but also occurs in Figures 9 and 10. For about the first 20 problem instances, our method performs quite a lot better than it does on the latter 25 problems. For GPG we can see the converse: GPG is slower on those first 20 instances than it is on the others. Again, this can be explained from the specific instances. In those first 20 problems, we have a number of additional packages that have to be transported. In our system, that means that all of the plan can remain intact, and a small planning problem is to be solved to reach the additional goals. Due to the intricacies of GPG, however, it removes quite a few actions from the plan before it is extended.

When we compare the results of the two replanning sys-

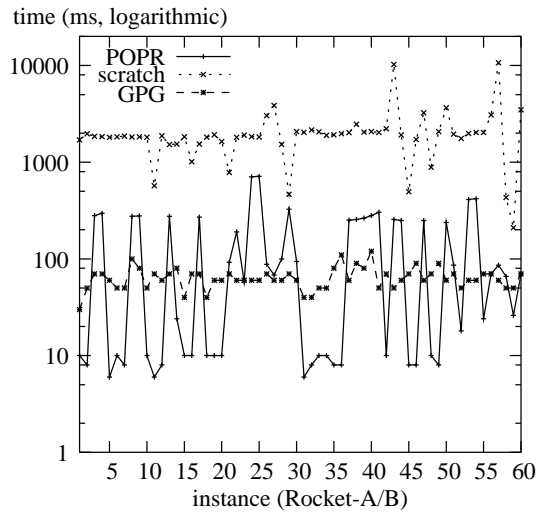


Figure 7: The run time for “Rocket”

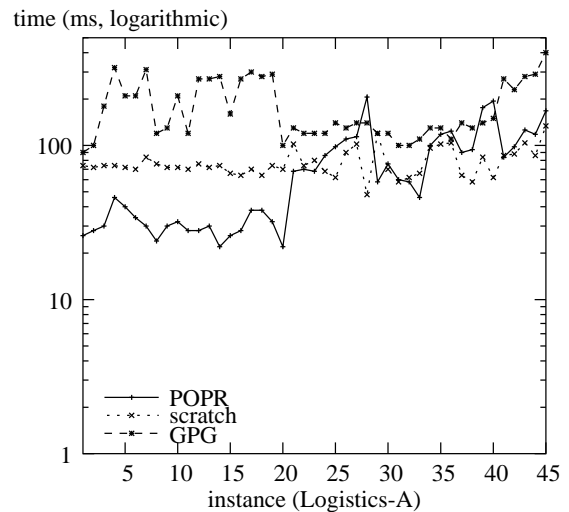


Figure 8: The run time for “Logistics-A”

tems, POPR and GPG, at a higher level, the differences are not directly clear from the graphs. However, Table 1 shows the means for the different problem sets, and the results of the analysis of variance (ANOVA). From these results it can be seen that the distinction between the two approaches is quite clear. We see that our plan repair method outperforms GPG in all domains except for the Rocket domain, where it is much slower in about half of the cases, resulting in a higher average of run time. We intend to perform a more thorough analysis of the results, based on the statistical methods discussed by Long and Fox (2003), in the near future.

The quality of the plans, when measured in number of actions, is slightly less when using plan repair. The reason for this is that it is sometimes easier to repair a plan without first removing redundant actions in the unrefinement phase, than

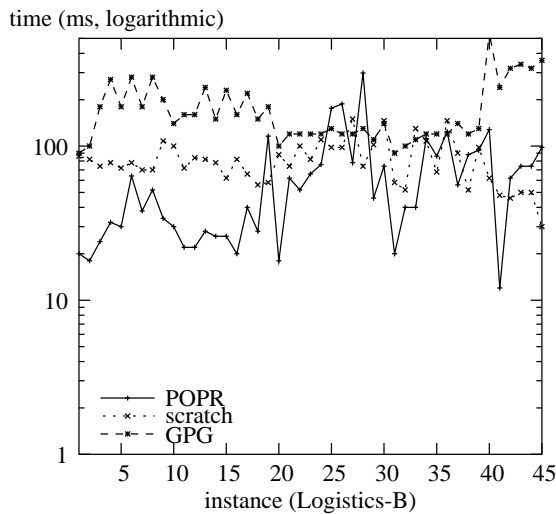


Figure 9: The run time for “Logistics-B”

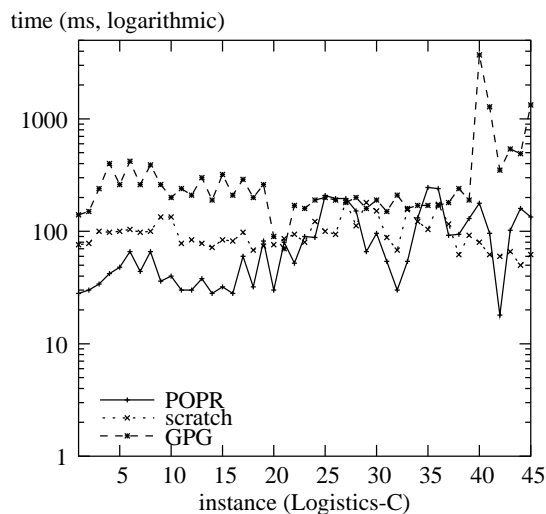


Figure 10: The run time for “Logistics-C”

it is to repair a plan that has redundant actions removed. The (estimated) ease with which a plan can be extended is used when deciding which plan to hand over to the refinement phase. Therefore, sometimes a quicker solution using more actions is chosen. This behaviour can for example be seen in the gripper domain: suppose that a ball has to be moved from location A to location C instead of to location B (a change in goals). When the robot ends its plan in location B, three actions are required to repair the plan: `pickup` the ball in B, `move` to C and `drop` the ball there. This requires a total of six actions to bring the ball to its final location: three to bring the ball to B (its original destination), and another three to bring it to C. Now, suppose that we would first unrefine the plan, and remove the three actions that bring the ball to B. This requires four actions to repair the plan: `move` to A, `pickup` the ball there, `move` to C and `drop` it. Therefore,

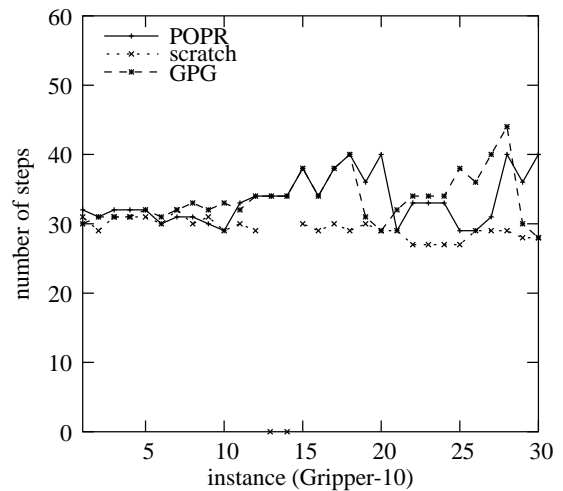


Figure 11: Size of the plans produced by planning from scratch and by using plan repair.

when this option is considered, it is deemed less favourable than the first option, since the resulting planning problem is estimated to be harder. Figure 11 shows the size of the resulting plans for the gripper domain. As we can see, the repaired plans are slightly larger than the plans computed from scratch. (Note that VHPOP cannot find a solution for instances 13 and 14, which is why they are missing.)

Related Plan Repair Methods

The idea of reusing an existing plan instead of planning from scratch is not novel. In this section, we discuss a number of existing approaches. First, some are based on Graphplan, like GPG (Gerevini & Serina 2000). SPA (Hanks & Weld 1995) uses a kind of local search starting with the original plan. Third, we found one that uses the structure of hierarchical task networks (HTN) to determine which actions to remove from a failed plan, called *Replan* (Boella & Damiano 2002). Furthermore, there are two methods that use plan repair rules: *Chef* (Hammond 1990), and *O-Plan* (Drabble, Dalton, & Tate 1997). Finally, we also found a couple of plan repair methods that rely not so much on classical AI planning techniques, but use somewhat more unrelated techniques, such as LPA* path planning by *Sherpa* (Koenig, Likhachev, & Furcy 2002), and a proof system by *MRL* (Koebler 1994). We shortly discuss each of these approaches and relate them to our plan repair template.

We first look at GPG, which we used as a comparison in our experimental results. It uses an approach based on the Graphplan planner (Blum & Furst 1997). Once a plan becomes invalid, GPG checks where inconsistencies occur in the plan. The plan is then divided into three parts: the *head* of the plan that consists of actions that can all be executed from the initial state, a middle part consisting of all inconsistent actions and the actions between, and a *tail* that can be used to attain the goals once the inconsistencies have been solved. These three parts can be identified using the planning graph that was constructed during the planning phase.

The middle part is then discarded (unrefinement, step 3.1) and a plan is sought to bridge the gap that exists between the head and the tail of the plan (refinement, step 4.1). If such a plan cannot be found, the gap is enlarged and the process repeats. Eventually, all of the plan will be discarded, in which case a completely new plan is constructed (if possible). This system does not explicitly make use of a history. Instead, it unrefines only the initial plan, and never any of the plans produced by a refinement step. This can be seen as an implicit memory.

The SPA planner (Hanks & Weld 1995) is another example showing the two parts of plan repair. It selects the next partial plan to work on (step 5) using a queue (implementing a breadth-first search in the space of plans). The partial plans on this queue are either to be refined (denoted by \downarrow), or to be unrefined (denoted by \uparrow). Either step 3.1 or 4.1 is chosen accordingly. For a partial plan tagged with \downarrow we derive all refinements, and add those to the queue. For a partial plan P tagged with a \uparrow , one decision made during planning is reversed (unrefinement), and next not only the resulting plan P' is added to the queue (again with a \uparrow), but also the refinements of this plan P' are added (with a \downarrow), except for the plan P . Tagging the plans in the queue with either \uparrow or \downarrow ensures that the same node in the search space is not visited twice, hence it can be considered a form of memory.

The Replan (Boella & Damiano 2002) model of plans is similar to the plans used in the hierarchical task network (HTN) formalism (Erol, Hendler, & Nau 1994). A task network is a description of a possible way to fulfill a task by doing some subtasks, or, eventually (primitive) actions. For each task at least one such a task network exists. A plan is created by choosing the right task networks for each chosen (abstract) task, until each network consists of only (primitive) actions. Throughout this planning process, Replan constructs a *derivation tree* that includes all chosen tasks, and shows how a plan has been derived.

Plan repair within Replan is called *partialisation*. For each invalidated leaf node of the derivation tree, the (smallest) subtree that contains this node is removed (unrefinement, step 3.1 of Algorithm 1). Initially, such an invalid leaf node is a primitive action, and the root of corresponding subtree is the task which network contained this action. Subsequently, a new refinement is generated for this task (step 4.1). If the refinement fails, a new round is started in which subtrees for tasks higher in the hierarchy are removed and regenerated. In the worst case, this process continues until the whole derivation tree is discarded. Like GPG, Replan never unrefines a plan produced by a refinement step, except for the initial plan. Again, this can be seen as implicitly using a memory.

Case-based planners have long since employed plan repair strategies to adapt their plans to new situations. One of the first approaches to the plan repair problem in case-based planning was the Chef system by Hammond (1990), a domain specific planning system for cooking. The Chef system is equipped with a set of *plan repair rules*. Each such rule describes how a specific failure can be repaired. When Chef encounters a failure, it builds an explanation of

why the failure has occurred. This explanation includes a description of the steps and states leading towards the failure as well as the goals that these steps tried to realise. Based on the explanation, a set of plan repair strategies are selected and instantiated to the specific situation of the failure. After choosing the best of these instantiated repair strategies, Chef implements it. Note that such a repair strategy consists of the specific refinements and unrefinements that have to be performed.

The strategy of using plan repair rules is also followed by O-Plan (Drabble, Dalton, & Tate 1997). However, there is no such thing as an explanation of a failure. During execution, the system confirms the effects of every action. For each failing effect that is necessary for some other action to execute, additional actions, in the form of a repair plan, are added to the plan. These repair plans are prebuilt plans that can repair certain failure conditions. For example, the repair plans may include a plan for changing a flat tyre, or for replacing a broken engine. Whenever an erroneous condition is encountered, the execution of the plan is stopped and a repair plan is inserted and executed. After completion of the repair plan, execution of the regular plan resumes again. O-Plan does only *add* actions to repair failures. Therefore, it does neither employ unrefinements nor does it require a history. It is therefore also *incomplete*: not all failures can be recovered from. Wang and Chien (1997) describe how search can be added to O-Plan, to try and recover from failures for which no prebuilt repair strategy is available. However, they too do not consider removing actions from a plan to recover from failures, but merely find out which actions they have to re-execute.

Another case-based system is MRL (Koehler 1994), based on a proof system. Upon retrieving a plan from the library that is to be adapted to the new situation, it tries to use the retrieved plan as a “proof” to establish the goal conditions from the initial state. If this succeeds, the plan can be used without alteration. In the other case, it results in a failed proof from which refitting information can be extracted. Based on the failed proof, a plan skeleton is constructed by using a *modification strategy* (Koehler’s term for an unrefinement strategy). This strategy uses the failed proof to derive which parts of the proof (i.e. the plan) are useful, and which are not. Then, the useless parts are removed. Having computed this skeleton, a proof system is used as a refinement strategy to fill in the gaps.

Finally, we mention the Sherpa replanner (Koenig, Likhachev, & Furcy 2002). In contrast to the previous systems, Sherpa applies the unrefinement step only once. It uses the LPA* algorithm, which was actually designed for *path* plan repair, to back-track to a partial plan that has the same heuristic value as before the unexpected changes in the world. From there on only refinement steps are used (i.e. normal planning). Because of the unrefinement strategy and the single application thereof, Sherpa does not work on all plan repair problems, but only on problems in which actions have been removed from the domain description. The unrefinement step then simply consists of removing those actions that are no longer available.

From this short overview it becomes clear that existing systems for plan repair also distinguish between a refinement and an unrefinement phase, although this is not always shown explicitly. We therefore conclude that the plan repair template is indeed general enough to describe most plan repair systems.

Discussion

This paper started with two important observations. First, we really need plan repair methods that start with the original plan. The reason is not only because we believe that this is often more efficient in practice, but also because an agent may have made bookings and commitments to others which may be costly to undo. Second, existing planning techniques are more advanced than existing plan repair techniques. Therefore, we would benefit from a straightforward method to use planning techniques for plan repair.

The main message of this paper is good news: planning techniques require only a relatively simple extension to be able to do plan repair as well. We showed how this can be done, using a plan repair template that describes two different operations required for repairing a plan: (i) removing actions that are in the way of attaining a goal, and (ii) adding actions required for attaining goals. Adding actions is similar to planning, but, surprisingly, as we have showed for VHPOP, planning heuristics can also be used for removing actions!

Furthermore, we have shown that

- the plan repair template is general enough to explain most existing plan repair approaches, and that
- the plan repair template is useful for devising a new unrefinement heuristic based on existing planning heuristics, and that
- POPR, an implementation of this heuristic on top of VHPOP, performs slightly better than GPG, and that
- plan repair in practice is much better than planning from scratch in spite of the theoretical result by Nebel and Koehler (1995).

We believe that this (template) view on plan repair methods will help in designing even faster and more powerful ones, and specifically in using advanced planning techniques for plan repair. Besides the current strategy that focuses on action deletion, other types of unrefinement strategies might be developed, for example the removal of causal links or the relaxation of (timing) variables.

Currently, we are applying the presented plan repair technique in multi-agent systems. Suppose a situation where an agent asks help from others, because it is not able to complete its plan on its own. When another agent agrees, both agents have to “repair” their plans: the requesting agent needs to include the effects of the supplying agent, and the supplying agent needs to add actions for this additional (sub)goal. This is one additional example that shows the importance of plan repair. Finally, we also try to improve upon the results of our heuristic by studying the behaviour in the “Rocket” domain and by doing a more thorough analysis of the results.

Acknowledgements

The authors would like to thank Ivan Serina for putting his benchmark set and the GPG planning system at our disposal, as well as for his comments on an earlier version of this paper. We are also grateful for the helpful comments of the reviewers.

References

- Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.
- Boella, G., and Damiano, R. 2002. A replanning algorithm for a reactive agent architecture. In *Artificial Intelligence: Methodology, Systems, and Applications (LLNCS 2443)*, 183–192. Springer Verlag.
- Drabble, B.; Dalton, J.; and Tate, A. 1997. Repairing plans on the fly. In *Proc. of the NASA Workshop on Planning and Scheduling for Space, Oxnard, CA*.
- Erol, K.; Hendler, J.; and Nau, D. S. 1994. Semantics for hierarchical task network planning. Technical Report CS-TR-3239, UMIACS-TR-94-31, Computer Science, University of Maryland.
- Gerevini, A., and Serina, I. 2000. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proc. of the Fifth Int. Conf. on AI Planning Systems (AIPS-00)*, 112–121. Menlo Park, CA: AAAI Press.
- Glover, F., and Laguna, M. 1993. Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems*. Scientific Publications, Oxford.
- Hammond, K. J. 1990. Explaining and repairing plans that fail. *Artificial Intelligence* 45:173–228.
- Hanks, S., and Weld, D. 1995. A domain-independent algorithm for plan adaptation. *Journal of AI Research* 2:319–360.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of AI Research* 14:253–302.
- Kambhampati, S.; Knoblock, C. A.; and Yang, Q. 1995. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence* 76(1-2):167–238.
- Kambhampati, S. 1997. Refinement planning as a unifying framework for plan synthesis. *AI Magazine* 18(2):67–97.
- Koehler, J. 1994. Flexible plan reuse in a formal framework. In *Proc. of the 2nd European Workshop on Planning (EWSP-93)*, 171–184. Vadstena, Sweden: IOS Press.
- Koenig, S.; Likhachev, M.; and Furcy, D. 2002. Lifelong planning A*. Technical Report GIT-COGSCI-2002/2, Georgia Institute of Technology, Atlanta, Georgia.
- Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *JAIR* 20:1–59.
- Nebel, B., and Koehler, J. 1995. Plan reuse versus plan generation: a complexity-theoretic perspective. *Artificial Intelligence* 76:427–454.
- van der Krogt, R.; de Weerd, M.; and Witteveen, C. 2003. A resource based framework for planning and replanning. *Web Intelligence and Agent Systems* 1(3/4):173–186.
- Wang, X., and Chien, S. 1997. Replanning using hierarchical task network and operator-based planning. Technical report, Jet Propulsion Laboratory Nasa.
- Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *JAIR* 20:405–430.