The two faces of plan repair

Roman van der Krogt and Mathijs de Weerdt

Delft University of Technology
{R.P.J.vanderKrogt,M.M.deWeerdt}@ewi.tudelft.nl

Abstract

Plan repair has two faces. Alternately, a plan repair method looks like a planning method, or looks like a method that does exactly the opposite, i.e., removing actions from a plan. We propose a general framework for plan repair that shows the relation between these two alternating steps. Any plan repair method has this property. This claim is supported by showing how a number of plan repair systems fit into the presented framework. One of the advantages of a general framework is that it helps to understand existing techniques and improve upon them. As an example of this, we present a novel heuristic for plan repair that can make use of existing planning heuristics. Some initial results are provided that indicate that this heuristic is competitive with existing plan repair methods.

1 Introduction

Planning is one of the most important methods to achieve the pro-active property that we would like agents to exhibit. In planning problems the agent has a description of that part of the world that is relevant, and a description of the goals it wants to attain. The task then is to find a (partially ordered) sequence of actions that brings the world from the current state into a state in which the agent has attained its goals. Often, when such a plan is executed, the world may change in an unexpected way; either because of actions by other agents or unexpected consequences of actions of the agent itself. When this happens, the agent needs to reconsider the remainder of its plan. This process is called *plan repair* (sometimes also referred to as *replanning*).

In previous work [10], we showed the similarities between such *plan repair* methods and normal planning methods: the so-called *refinement planning framework* [7] was modified to model plan repair methods as well. Such a general view on planning and plan repair methods not only helps to compare and understand existing algorithms and heuristics, but also to develop new methods.

Studying plan repair methods more closely, we discovered that within any plan repair method two distinct, but alternately used, heuristics can be recognized. One face of plan repair is shown when the heuristic is active that "decides" when and what part of a plan should be *removed*, because it (probably) hinders the reachability of the goal(s). The other face of plan repair is shown when the plan is *extended* in order to reach the goals. In that case, a plan repair method acts similarly to a "normal" planning method.

In this paper, we support this new view by showing how existing plan repair methods fit into a framework that models the two alternating steps in the search for a new valid plan. To this end, we first briefly recapitulate the refinement planning approach and show how it can be extended to an *unrefinement replanning* framework. The template algorithm of this framework clearly shows the two faces of plan repair. We proceed by discussing two plan repair systems, Replan [2] and SPA [6], showing how they can be conceived as instances of the presented framework. To show how the framework can help to develop new techniques, we present a new heuristic for breaking down plans, that makes use of existing heuristics for planning. We present some of our initial experiments using this heuristic, which indicate that the heuristic is competitive with existing algorithms. First, however, we briefly summarize the most important elements of the refinement framework for plan construction.

2 Refinement Planning

Plans are constructed to solve a certain *planning problem*. Such a planning problem, denoted by Π, is described by (i) information about which actions can be used, (ii) the description of the initial state, and (iii) a specification of the goal state.

A plan is a sequence of actions. 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* [7, 8]. Since most existing (classical) planning algorithms can be conceived in this way, it can be considered a *unifying view* on planning. The idea behind refinement planning is that we start with a 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") 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*.

The set of candidate plans that a partial plan P represents is denoted by *candidates*(P). This set contains all action sequences c in which all actions from P are present in an order consistent with the ordering described by the partial plan. Note that *candidates*(P) may include plans with *more* actions than P, as long as the constraints are all met. We define a *minimal candidate* to be a candidate that does not contain more actions than the partial plan 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 $\mathcal{P} = \{P_1, \ldots, P_n\}$, such that for each of the new partial plans, the candidate set is a subset of *candidates*(P). Furthermore, we introduce a function called *solution* that can be used to determine whether the minimal candidate of a partial plan is a solution to a given planning problem. If it is, the sequence of actions that solves the problem is returned.

A general refinement planner works 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.

By removing the restriction that we can only *add* constraints, refinement planning can be seen as a unifying view on both planning and plan repair [10]. However, it is not very elegant, and, more importantly, hides the fact that plan repair really constitutes two separate activities: removing actions 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 the *unrefinement planning* approach.

3 Plan Repair

In the previous section, we discussed Khambampati's [7] refinement planning as a unifying approach to planning. This refinement planning approach always *adds* constraints to the partial plan. However, to recover from errors, we may have to *remove* actions, or orderings or other constraints, to fix a plan. Thus, the refinement planning approach is not suitable for plan repair purposes. However, just as the refinement planning approach provides a template for planning algorithms, we would like to have a template for plan repair algorithms.

The 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. Refinement takes place as in the regular refinement planning approach (step 4.1 in Algorithm 1). Second, we use a *history* $\mathcal{H}$ to keep track of the refinements and unrefinements we 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 [5] may be employed to best make use of this available memory. The refinement plan repair template is depicted in Algorithm 1.

4 Existing Strategies

In this section we support our claim that the plan repair template is a unifying approach to plan repair by showing how some existing plan repair algorithms can be conceived as instances of the template algorithm (Algorithm 1). More specifically, we show which (un)refinement strategies are present in these systems.

The first system we look at is called Replan [2]. Their model of plans is similar to the plans used in the hierarchical task network (HTN) formalism [3]. 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

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

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, Π) *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$(P_i, \Pi, \mathcal{H}')$.

end

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 leave node of the derivation tree, the (smallest) subtree that contains this node is removed (unrefinement, step 3.1). Initially, such an invalid leave 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.

The SPA planner [6] is another example showing the two faces 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 considered twice. This way SPA does not explicitly need a history.

In many other plan repair methods the same distinction between refinement and unrefinement strategies can be made. For example, in MRL [9] the refinement strategy employs a proof system to complete the plans. If it fails, the unrefinement strategy removes ineffectual actions. GPG [4] uses a similar method, but uses a refinement planning part like Graphplan [1].

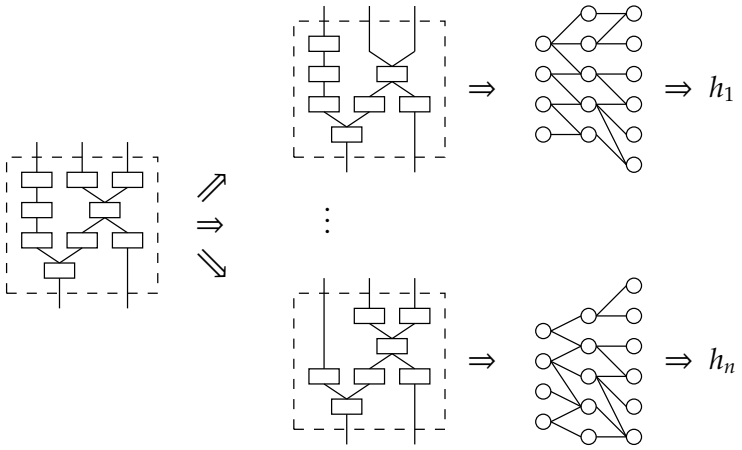

Figure 1: Sketch of the unrefinement heuristic. From the original plan on the left, we derive n subplans and calculate heuristic values ($h_1, \ldots, h_n$) for them using (in this case) a planning graph heuristic.

5 A New Unrefinement Heuristic

Besides a unifying view on replanning systems, the template algorithm also gives us pointers for devising new plan repair methods. In the methods presented in the previous section, the refinement and unrefinement strategies are tuned such that they complement each other. In this section we present an unrefinement heuristic 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 are automatically tuned, because the unrefinement heuristic makes use of the refinement heuristic to calculate heuristic values. This means that using our method, we can add plan repair capabilities to most existing planners. This has the additional benefit that our method can be easily upgraded when new and more efficient planning heuristics are devised.

Our approach to unrefinement 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. For each of the resulting plans, we use the chosen planning heuristic (for example, a planning graph heuristic) to estimate the amount of work it will require to transform this plan into a valid plan (i.e. a heuristic value for that plan is calculated). The plan that has the best heuristic value is selected and a refinement strategy is used to complete this plan. The steps that this procedure consists of are now discussed in greater detail.

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. Considering an exponential number of plans is no option for a fast heuristic. Instead, we only consider removing certain sets of actions. These sets have the following requirements: firstly, the actions should form a tree of a number of levels deep. At the first level, we have exactly one action, subsequent levels should either consist of all actions that satisfy preconditions of the actions on the previous level (if the tree is going *backwards* in time), or it should consist of all actions that have preconditions satisfied by actions at the previous level (if the tree is *forward* in time). Secondly, the root action of the tree should be an action at the beginning of the plan (if the tree is forwards), or at the end of the plan (if the tree is backwards). We call such sets of actions *removal trees*. Figure 2 shows an example of a removal tree of three levels (shown in grey). Each square represents an action, an arrow between two actions indicates that the first action supports a precondition of the second one. The number of removal trees in a given plan P is polynomial in the size of P.

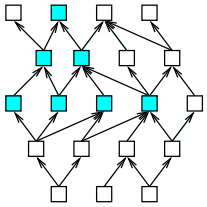

Figure 2: A backward removal tree

Given a removal tree, the second step is to calculate the heuristic value for that plan. To do this, we construct the plan that results when removing the removal tree. Next, we can simply apply the selected planning heuristic to obtain a heuristic value for the plan. Some heuristics have a problem with calculating a heuristic value for the kind of broken down plans we produce. To overcome this problem, we can construct a special domain. This domain consists of the original domain, as well as special actions encoding the plan that we would like to reuse. For this purpose, the plan is broken down into separate parts, called *cuts* (as shown in Figure 3). Each cut is chosen such that there are no two actions in a cut that were previously connected through one or more removed actions. (This is the reason that in Figure 3, the two larger cuts are separated.) 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.

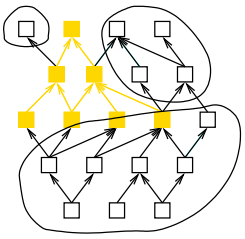

Figure 3: Example cuts of the resulting plan

The complete unrefinement strategy now works as follows: we begin by removing removal trees of depth one. If the heuristic reports that one or more of the plans can be expanded to a valid plan, we use the refinement strategy to try and complete those plans. If a valid refinement cannot be found, we iteratively increment the depth of the removal trees and try again, until we find a solution (or fail).

For the experimental validation of our technique, we integrated it into the VHPOP planner [11]. Experiments were performed using the benchmark set of GPG [4]. This benchmark set 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. Figure 4 shows some of our initial results. The

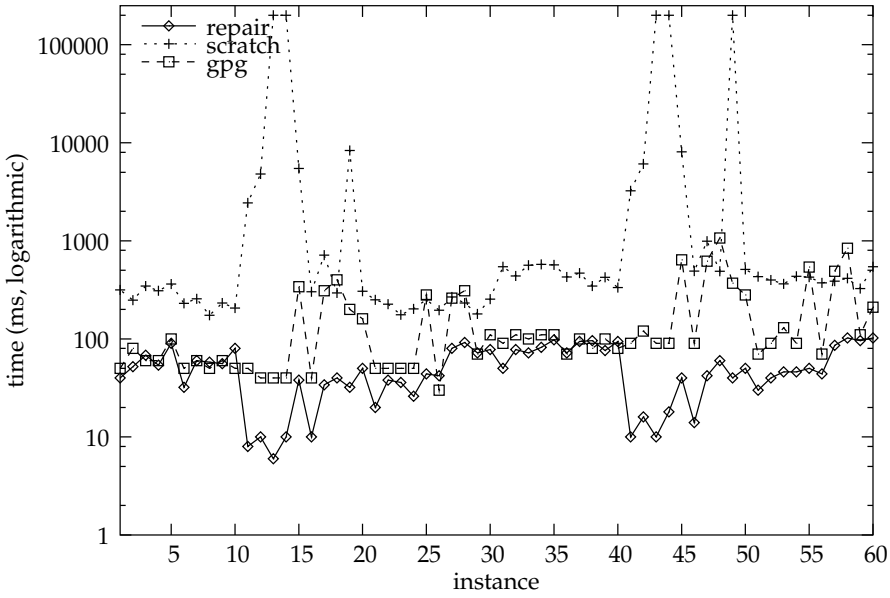

Figure 4: Runtime for the 60 problems in the Gripper domain. The first 30 problems are with 10 balls, problems 31-60 have 12 balls.

graph plots the running times for 60 instances of *gripper* problems. In the *gripper* domain, a robot equipped with two grippers can move through a number of rooms, and has to move balls from their current location to another. Each instance of a problem consists of a modification to a base problem. Problems 1-30 all use the same base problem with 10 balls, problems 31-60 use a problem with 12 balls. Examples of modifications 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". Between one and eight of such modifications are made to obtain a new problem instance. The graph plots the run-times for planning from scratch (using VHPOP , labeled *scratch* in the graph), for our plan repair method (labeled *repair*) and for the GPG plan adaptation system (labeled *gpg*). As one can see, plan repair is usually faster than solving a problem from scratch. Of the two plan repair systems, ours is faster than GPG for most of the problems, and only slightly slower on the others. The other problem sets in the benchmark show similar results.

6 Discussion

When a plan becomes invalid, a new plan needs to be searched for. Searching a plan from scratch starts with the set of all possible plans according to the refinement planning template. However, during plan repair we would like to stay as close as possible to the original plan. Therefore, it may be more efficient to

start the search from the existing, invalid, plan. In this paper, we claimed that in each existing plan repair algorithm two different types of heuristics are used. One of these is similar to planning heuristics, but the other is quite different; it decides whether to remove some part of the plan, and if so, which part to remove. We showed that existing systems indeed fit into a plan repair framework in which a clear distinction is made between these two types of heuristics.

In the previous section, we showed how such a framework can help in developing a new plan repair algorithm. In particular, we developed an general unrefinement heuristic that can be used in conjunction with existing planning heuristics. We briefly showed that this heuristic is competitive with existing plan repair methods, and planning from scratch.

We are currently performing more experiments with the new heuristic. Also, we are studying the effect of a plan library on plan repair. The idea is that the search can be sped up using small plan parts that are available. Furthermore, we would like to explore this idea in the context of multi-agent plan repair, in which other agents may be able to support the plan repair of one agent.

References

[1] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[2] G. Boella and R. Damiano. A replanning algorithm for a reactive agent architecture. In *Artificial Intelligence: Methodology, Systems, and Applications (LLNCS 2443)*, pages 183–192. Springer Verlag, 2002.

[3] K. Erol, J. Hendler, and D. S. Nau. Semantics for hierarchical task network planning. Technical Report CS-TR-3239, UMIACS-TR-94-31, Computer Science, University of Maryland, 1994.

[4] A. Gerevini and I. Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proc. of the Fifth Int. Conf. on AI Planning Systems (AIPS-00)*, pages 112–121, Menlo Park, CA, 2000. AAAI Press.

[5] F. Glover and M. Laguna. Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems*. Scientific Publications, Oxford, 1993.

[6] S. Hanks and D. Weld. A domain-independent algorithm for plan adaptation. *Journal of AI Research*, 2:319–360, 1995.

[7] S. Kambhampati. Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, 18(2):67–97, 1997.

[8] S. Kambhampati, C. A. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 76(1-2):167–238, 1995.

[9] J. Koehler. Flexible plan reuse in a formal framework. In *Proc. of the 2nd European Workshop on Planning (EWSP-93)*, pages 171–184, Vadstena, Sweden, 1994. IOS Press.

[10] R. van der Krogt, M. de Weerdt, and C. Witteveen. A resource based framework for planning and replanning. *Web Intelligence and Agent Systems*, 1(3/4):173–186, 2003.

[11] H. L. S. Younes and R. G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of AI Research*, 20:405–430, 2003.