Unrefinement Planning: Extending Refinement Planners with Plan Repair Capabilities

Roman van der Krogt
Delft University of Technology
r.p.j.vanderkrogt@ewi.tudelft.nl

Abstract

Kambhampati's *refinement planning* framework provides a unifying view on planning algorithms. It shows how a planning problem can be solved by successively applying a refinement operator, which adds more and more detail to the plan. In this paper, we show how this framework can be extended to include plan repair capabilities. This extension relies on the inclusion of *unrefinement* operators, that remove obsolete or obstructing actions from the plan.

Based on this general plan repair framework, we develop a generic plan repair heuristic. This heuristic consists of a new unrefinement strategy, and it can make use of existing planning heuristics as the refinement operator. The unrefinement strategy deteriorates a plan in a number of ways, by removing certain combinations of actions. The planning heuristic is then used to find the most promising candidate, which is then given to the planner as the initial partial plan to refine. The big advantage of the heuristic is that it allows to employ existing planning heuristics. This way, it can not only make use of advances in planning technology, but it can also be used to extend existing planners, giving them plan repair capabilities.

The performance of the heuristic is demonstrated by adapting an existing planner, and comparing the results with planning from scratch and a plan adaptation system. Experiments are presented that show that the new heuristic is very competitive with the other systems.

1 Introduction

Often, when 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. It can do this by planning from scratch again, but the agent can also try and modify the existing plan. This latter process is called *plan repair* (sometimes also referred to as *replanning* or *plan adaptation*). In theory, plan repair is no more efficient than a complete replanning from scratch (Nebel and Koehler, 1995). However, we expect that in practice, plan repair should often be more efficient, since a large part of the plan is usually still valid. Therefore, our goal is to develop an efficient plan repair technique that performs well in such cases.

In previous work (van der Krogt et al., 2003), we showed the similarities between plan repair and normal planning methods: the so-called *refinement planning framework* (Kambhampati, 1997) was modified to model plan repair methods as well. The proposed extension, however, was in some ways against the core ideas of the refinement planning approach. In particular, it lifted the restriction that refinement strategies always add constraints to the partial plan under consideration. This observation lead us to reconsider our approach. The result of this reconsideration is what we call the *unrefinement planning* framework. This framework clearly separates the two tasks that form plan repair: one task "decides" when and what part of a plan should be *removed*, because it (probably) hinders the reachability of the goal(s). The other task of plan repair is when the partial plan is subsequently *extended* in order to reach the goals. This latter task is performed by regular refinement operators. The former task requires special *unrefinement operators*.

Using this unrefinement planning framework, we can develop new heuristics for plan repair. One such heuristic is presented in this paper. It consists of a generic unrefinement strategy, and allows existing planning systems to be used as the refinement strategy. The unrefinement strategy generates a number of unrefinements of the plan (i.e. ways to remove actions from it). These unrefinements are then evaluated using the heuristic of the chosen refinement strategy. The most promising is then given to the planner to finish it. The benefit of this approach is twofold: on the one hand, this allows us to concentrate on developing an efficient unrefinement strategy, using existing planners as a refinement strategy. On the other hand, it allows us to extend existing planners with plan repair capabilities, making them more applicable to real-world situations.

In the remainder of this paper, we first discuss the unrefinement planning framework. We discuss the differences with refinement planning, and give an indication of how existing systems can be seen to fit into the template. Then, we introduce our new plan repair method, showing how we decide which actions to remove from a plan, and how the planning heuristic is employed to evaluate the options. Finally, we present experimental results showing that our approach can be considered competitive with existing approaches.

2 Unrefinement 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 et al., 1995; Kambhampati, 1997). 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 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") 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 $\mathcal{P} = \{P_1, \ldots, P_n\}$, such that for each of the new partial plans, the candidate set is a subset of *candidates*(P). 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 (van der Krogt et al., 2003). However, it is not very elegant, and, more importantly, it 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.

The main idea behind the unrefinement planning approach, 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 actions 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 plan is extended (refined) to satisfy the goals. For example, consider a plan for attending this years PlanSIG, given a certain budget. This probably consists of actions such as `book-flight`, `book-accommodation`, `travel-to-airport`, etc and requires most of the budget. Now suppose that the PlanSIG is moved to, say, Paris. Since the current plan leaves no room on the budget to further refine the plan to satisfy the new goals (e.g. by booking another flight from Cork to Paris), we have to remove actions from the plan that are interfering with a possible extension. In this case, the actions

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

booking a flight and accommodation have to be removed, freeing budget to plan for the new trip. Thus, besides a *refinement* strategy extending a plan, the planner should also employ an *unrefinement* strategy for retracting constraints from the partial plan.

An extension of the refinement planning template algorithm that allows for unrefinement strategies to be employed, can be found in Algorithm 1. 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 (step 3.1). Refinement takes place as in the regular refinement planning approach (step 4.1). 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 (Glover and Laguna, 1993) may be employed to best make use of this available memory.

Existing plan repair and plan adaptation systems can be shown to fit within this framework. Examples of these systems include Replan (Boella and Damiano, 2002), SPA (Hanks and Weld, 1995), GPG (Gerevini and Serina, 2000), and Sherpa (Koenig et al., 2002). As a specific example, we look at GPG, which we have used as a comparison in our experimental results. It uses an approach based on the Graphplan planner (Blum and Furst, 1997). Once a plan is 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).

3 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 existing methods (such as mentioned 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

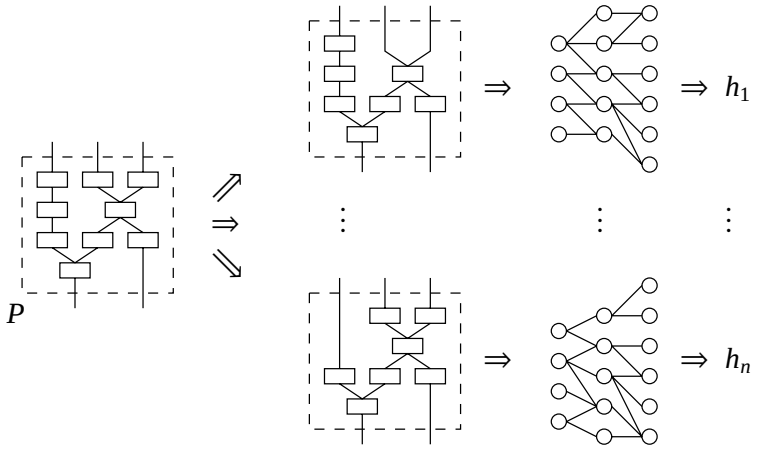

Figure 1: Sketch of the unrefinement heuristic. From the original plan P on the left, we derive n subplans and calculate heuristic values ($h_1, \ldots, h_n$) using (in this case) a planning graph heuristic.

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, which is clearly too much to consider all. Therefore, we only consider removing certain sets of actions, focusing on actions that are either depending on the initial state, or actions that produce unused positive effects. The idea is that these actions are on the borders of the plan, and that by removing them, we shrink the plan from the outside in. More specifically, 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 will 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 an action producing an unused effect. The *height* of the tree determines which actions are selected in the graph. The following rules determine the actions in a removal tree:

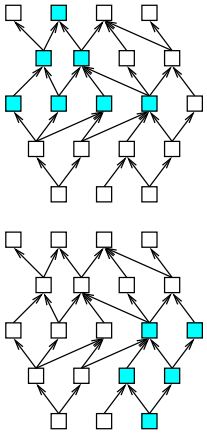

Figure 2: A backward removal tree (top) and a forward tree (bottom).

1. For an action o, either depending on the initial state, or producing an unused 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 unused effects, 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 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 that root (indirectly) depends upon. The tree at the bottom is an upward removal tree, containing actions depending upon the root action.

If we would only consider 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 plans 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 a single one, consisting of the actions in T_1, T_2 and T_3. For an example, consider Figure 3. This figure shows the result of merging the two trees shown in Figure 2.[1] 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. The set of merged trees is used to calculate possible unrefinements to the plan, as discussed next.

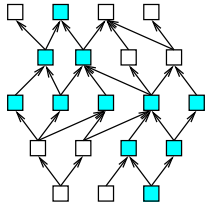

Figure 3: The merger of the trees of Figure 2.

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 removing the removal tree. Next, we can simply apply the selected 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.[2] 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*. 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 4 shows the cuts of the plan that results from removing the removal tree of Figure 2(top).

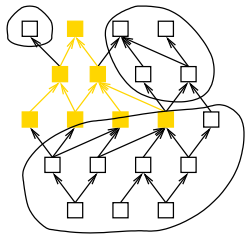

Figure 4: Example cuts of a plan.

The complete unrefinement strategy now works as follows: we begin by computing the removing removal trees of depth one. Overlapping removal trees are merged as discussed before, and for each 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

[1] Notice 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.

[2] For example, forward heuristics (such as used in e.g. FF (Hoffmann and 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.

trees and try again. This procedure is repeated, until, finally, the whole plan is discarded and a complete replanning is performed.

4 Experimental Results

For the experimental validation of our technique, we integrated it into the VHPOP planner (Younes and Simmons, 2003). This planner was chosen since it is a clear refinement planner, that sticks close to the original template algorithm. This makes it easier to add our extensions.[3] Experiments were performed using the benchmark set of GPG (Gerevini and Serina, 2000). This benchmark set is the only one for plan repair problems that the author is 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".

Figure 5 shows 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 our version of VHPOP using the proposed plan repair method (labeled *repair*) 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 Figure 5 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 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 when performing plan repair. When we compare the results of the two replanning systems, we see that GPG is the slowest performer on the *logistics* problems; there, it is even outperformed by planning from scratch in most cases. In the other domains, GPG and our method differ not much.

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 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 it 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, when this option is considered, it is deemed less favourable than the

[3] As an indication of how hard (or easy) it is to adapt an existing planner using this framework, we observe that the source code of VHPOP consists of 21,080 lines of code, whereas the adapted system consists of 23,757 lines.

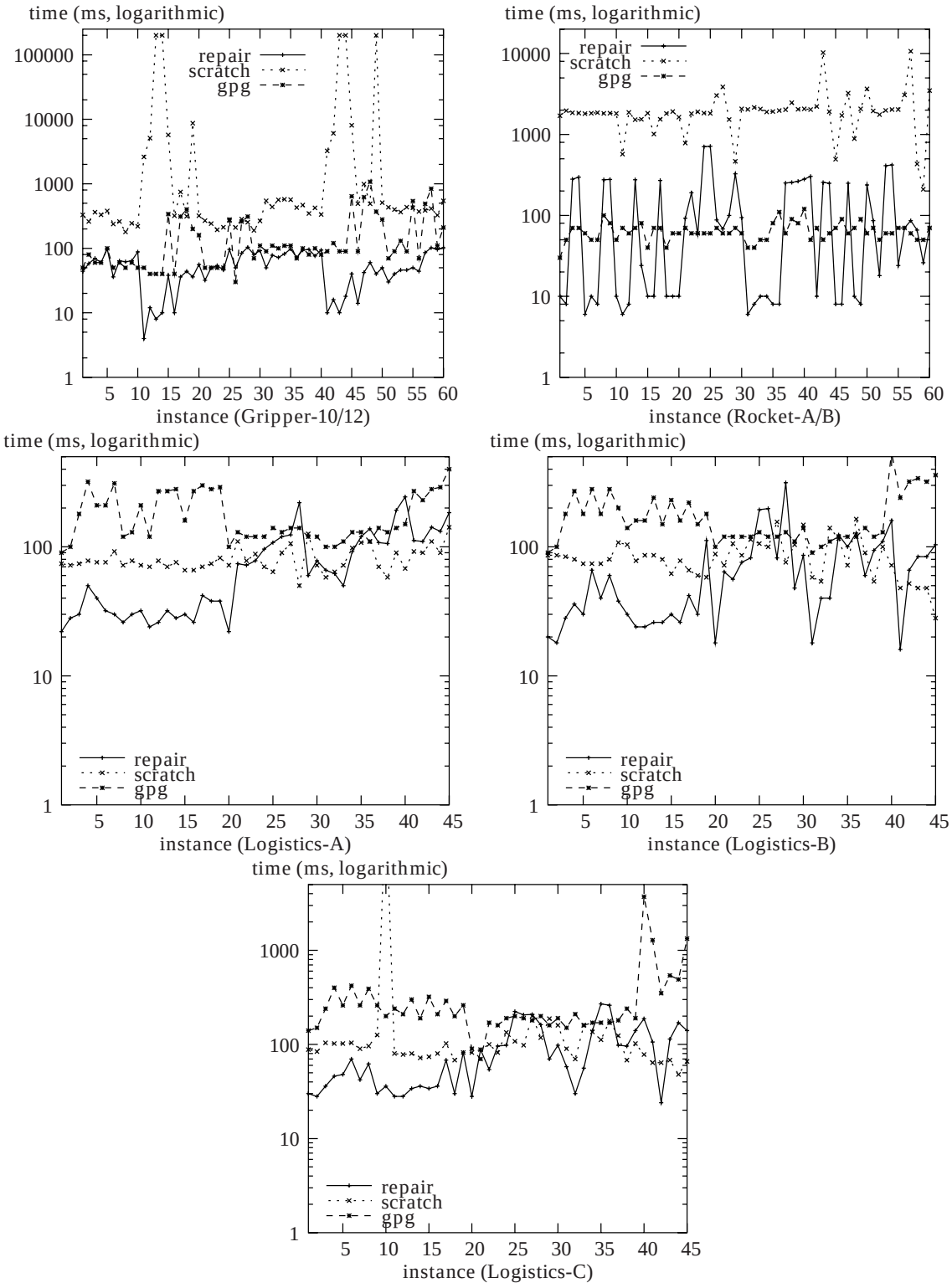

Figure 5: CPU seconds (logarithmic scale) required by VHPOP from scratch, VHPOP using plan repair and GPG for the GPG benchmark set.

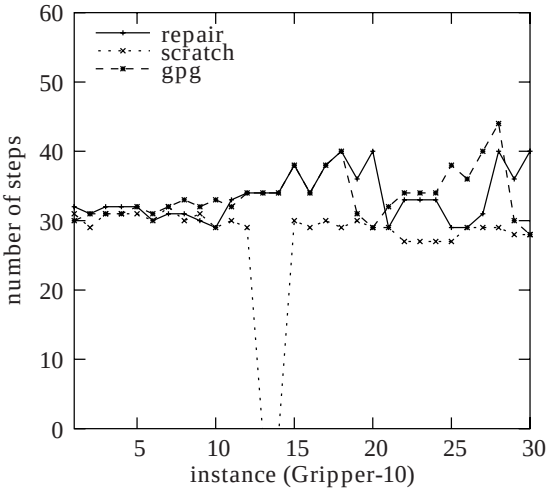

Figure 6: Size of the plans produced by planning from scratch and by using plan repair.

first option, since the resulting planning problem is estimated to be harder. Figure 6 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.

5 Discussion

Kambhampati's refinement planning framework (1997) can be considered a unifying view on planning algorithms. It provides a template algorithm, of which existing planning algorithms can be shown to be an instantiation. In real world systems, however, a planning algorithm does not suffice: we should be able to deal with *plan repair* in the case of unforeseen events. In this article, we showed how we can extend the refinement planning approach with plan repair capabilities. The resulting framework is called the *unrefinement planning* approach.

We used the ideas provided by this framework, to develop a new unrefinement strategy. This strategy was developed such that it is able to use existing planning heuristics in its computation. It works by calculating so-called *removal trees*, and using the planning heuristic to estimate the amount of work it takes for the planner to extend the resulting plans into complete plans. The benefit of this is that we can use this strategy to add plan repair capabilities to existing planners. Furthermore, we can easily incorporate advances in planning systems.

The new heuristic was used to support plan repair in the VHPOP planner (Younes and Simmons, 2003). Experimental results show that the heuristic outperforms planning from scratch and is competitive with GPG (Gerevini and Serina, 2000), performing similarly or better (depending on the domain). The experiments also confirm the fact that plan repair can lead to less efficient plans. This is hardly avoidable when repairing plans, as it results from trying to reuse the existing plan.

We are currently performing more experiments with the new heuristic and variants of it. 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. We are also interested in adapting other planning systems using this approach. 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.

Acknowledgements

Roman van der Krogt is supported by the Freight Transport Automation and Multi-Modality (FTAM) program of the TRAIL research school for Transport, Infrastructure and Logistics. The author wishes to thank Mathijs de Weerdt for his comments and help.

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)*, pages 183–192. Springer Verlag.

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)*, pages 112–121. AAAI Press, Menlo Park, CA.

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

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

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.

Koenig, S., Likhachev, M., and Furcy, D. (2002). Lifelong planning A*. Technical Report GIT-COGSCI-2002/2, Georgia Institute of Technology, Atlanta, Georgia.

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 Weerdt, M., and Witteveen, C. (2003). A resource based framework for planning and replanning. *Web Intelligence and Agent Systems*, 1(3/4):173–186.

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