

## **Plan Repair**

**A framework and a new heuristic with applications to logistics**

**TRAIL Research School, Delft, November 2004**

### **Authors**

**Ir. Roman van der Krogt**

**Dr. Mathijs de Weerd**

Faculty of Electrical Engineering, Mathematics and Computer Science, Delft  
University of Technology

© 2004 by R. van der Krogt, M. de Weerd and TRAIL Research School



# Contents

## Abstract

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
<b>2</b>	<b>Refinement Planning.....</b>	<b>3</b>
<b>3</b>	<b>Plan repair .....</b>	<b>5</b>
<b>4</b>	<b>Existing Strategies .....</b>	<b>6</b>
<b>5</b>	<b>A New Unrefinement Heuristic.....</b>	<b>7</b>
<b>6</b>	<b>Experimental Results .....</b>	<b>9</b>
<b>7</b>	<b>Discussion .....</b>	<b>13</b>
	<b>Acknowledgements .....</b>	<b>13</b>
	<b>References.....</b>	<b>14</b>



## Abstract

Planning can be a valuable tool for supporting a wide array of real-world problems, such as logistics, manufacturing and control. However, these applications are often highly dynamic, resulting in plans that require updating. In such situations, *plan repair* methods can be used to adapt the plan.

In this paper, we propose a general framework for plan repair. This framework is based on an existing general framework for planning, the so-called *refinement planning* approach. 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 show how we can extend an existing planning method into a system that can also deal with plan repair problems. This system is tested on a number of benchmark problems that deal with abstract transportation problems.

## Keywords

planning, replanning, transport, logistics



# 1 Introduction

In our everyday lives, we are often confronted with problems of the form “How should I achieve  $X$ ?”. For example,  $X$  can be things like “finishing this research paper”, “traveling to Edinburgh” or “finding information on the TRAIL research school”. The solution to such problems often take the form of a list of steps to take. Problems of this type are referred to as *planning problems*. More formally, in a typical planning problem, the agent (the person or entity that wishes to achieve a certain goal) has a description of that part of the world that is relevant (the *current state* of the world), a description of the *goals* it wants to attain, and the list of steps (often referred to as *actions*) that may be taken (including their preconditions and effects). The task, then, is to find a (partially ordered) sequence of actions that (when executed by the agent) brings the world from the current state into a state in which the agent has attained its goals.

*Example 1. Suppose our desire is to travel to Edinburgh. To model this as a planning problem, we should specify the current state, the goal, and the actions that we may take.*

- *The specification of the current state should obviously contain our current location. Furthermore, it should specify train and bus timetables, flight schedules, taxi fares, and all other information that may be required during planning.*
- *The goal in this case is easy, we want our location to be in Edinburgh.*
- *The list of steps that we can take includes such actions as “booking a flight”, “buying a ticket”, “biking to the train station”, “ringing a cab”, “checking-in for a flight”, etc. For all of these, we list the necessary preconditions and effects. For example, “checking-in for a flight” requires a reservation and our presence at the right check-in desk (with any baggage that we want to bring). The effect of our action would be that our luggage disappears into the dark vaults of Schiphol (hopefully to reappear in Edinburgh), and that a boarding pass has been issued to us.*

*A plan for this problem could start as follows: “book a flight”, “walk to the station”, “buy a train ticket”, “travel to Schiphol Airport”, “proceed to the right check-in desk”, etc.*

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*).

*Example 2. An example of failures in our plan for traveling to Edinburgh could be an action that fails, such as missing the train (e.g. due to a long queue at the ticket counter). Also the initial state or goals may change and render a plan broken. For example, our goal may change to be in a specific location in Edinburgh, instead of Edinburgh in*

*general, or the initial state may change in that there are no trains going to Schiphol this early in the morning, because of construction works.*

In previous work (van der Krogt et al., 2003), we showed the similarities between such *plan repair* methods and normal planning methods: the so-called *refinement planning framework* (Kambhampati, 1997) was shown to be able to model plan repair methods as well, by relaxing certain restrictions to it. Subsequent work (first reported in (van der Krogt and de Weerd, 2004)) led to the *unrefinement planning framework*, which is a general framework to model plan repair systems. There are two clear benefits of the new framework with respect to the (ab)use of the refinement planning approach as we did earlier:

1. It is clearer, more precise and more elegant than our initial attempt at using the refinement planning framework as a basis for a plan repair framework.
2. The new framework clearly distinguishes the two alternating phases of plan repair: on the one hand, plans need to be broken down to some extent (to remove actions that are no longer applicable in the new situation) and on the other hand, plans need to be extended to include actions solving the problem.

A benefit of general frameworks is that it allows to compare existing methods and devise new ones based on the framework. In this paper, we discuss the unrefinement planning framework and briefly show how some existing plan repair systems can be conceived as instances of the framework. To show how the framework can help to develop new techniques, we present a heuristic method that can be used to incorporate existing planners with plan repair. Experimental results show that the resulting system is competitive on a number of benchmark problems in some abstract transportation domains.

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  $\Pi$ , 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. It can be constructed in a series of steps. One way to look at this construction process is to see it 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 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”, or “action 5 should take place before action 14”) until all plans that match the constraints are solutions to the planning problem.<sup>1</sup> 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  $c$  for which all the actions in  $P$  are present in  $c$ , 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  $\mathcal{P} = \{P_1, \dots, P_n\}$ , such that for each of the

---

<sup>1</sup>Note that the presence of a particular action is considered a constraint on the final plan as well: the constraints are not defined over actions, but over the plan as a whole.

---

### Algorithm 1 REFINE ( $\mathcal{P}, \Pi$ )

**Input:** A partial plan  $\mathcal{P}$  and a problem  $\Pi$

**Output:** A solution to  $\Pi$  or ‘fail’

**begin**

1. **if**  $candidates(\mathcal{P})$  is empty **then**
  - 1.1. **return** fail;
2. **if**  $solution(\mathcal{P}, \Pi)$  returns a solution  $\Delta$  **then**
  - 2.1. **return**  $\Delta$ ;
3. Select a refinement strategy  $\mathcal{R}$  and generate the new plan set  $\mathcal{P}' = \mathcal{R}(\mathcal{P})$ .
4. Non-deterministically select a component  $\mathcal{P}'_i \in \mathcal{P}'$  and call REFINE( $\mathcal{P}'_i, \Pi$ ).

**end**

---

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 uses an algorithm as outlined in Algorithm 1. 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.<sup>2</sup> Proceed until a solution is obtained, or the set of partial plans is empty. Clearly, the particular refinement strategy is crucial to this approach. It is therefore that different planning algorithms mainly differ in their refinement strategy.

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, 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.

---

<sup>2</sup>Note that the “select” here is in fact a non-deterministically selection. In practice, the wrong one may be selected. In that case, the process should be tracked back, and continued at this point with another decision.

### 3 Plan repair

In the previous section, we discussed Khambampati’s (1997) 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, as part of the process of repairing our 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.

Our 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 2). 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. The refinement plan repair template is depicted in Algorithm 2.

---

#### Algorithm 2 PLAN REPAIR ( $P, \Pi, \mathcal{H}$ )

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

**Output:** A solution to  $\Pi$  or ‘fail’

**begin**

1. **if**  $\text{candidates}(P)$  is empty **then**
  - 1.1. **return** fail;
2. **if**  $\text{solution}(P, \Pi)$  returns a solution  $\Delta$  **then**
  - 2.1. **return**  $\Delta$ ;
3. **if** we choose to unrefine **then**
  - 3.1. Select a unrefinement strategy  $\mathcal{D}$  and generate the new plan set  $\langle \mathcal{P}, \mathcal{H}' \rangle = \mathcal{D}(P, \mathcal{H})$ .
4. **else**
  - 4.1. Select a refinement strategy  $\mathcal{R}$  and generate the 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**

---

## 4 Existing Strategies

In this section we support our claim that the plan repair template is a unifying approach to plan repair. We do this by showing how some existing plan repair algorithms can be conceived as instances of the template algorithm (Algorithm 2). More specifically, we show which refinement and unrefinement strategies are present in these systems.

The first system we look at is called Replan (Boella and Damiano, 2002). Their model of plans is similar to the plans used in the hierarchical task network (HTN) formalism as described by Erol et al. (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). Initially, such an invalid leaf node is a primitive action, and the root of the corresponding subtree is the task whose 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 GPG system by Gerevini and Serina (2000) uses an approach based on the Graphplan planner (Blum and Furst, 1997). The idea of GPG is to divide a plan in three parts: the *head* of the plan that consists of actions that can all be executed from the current state, a *tail* that can be executed from some state during execution of the plan to obtain the goals, and a middle part consisting of actions coming *after* the head and *before* the tail. 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).

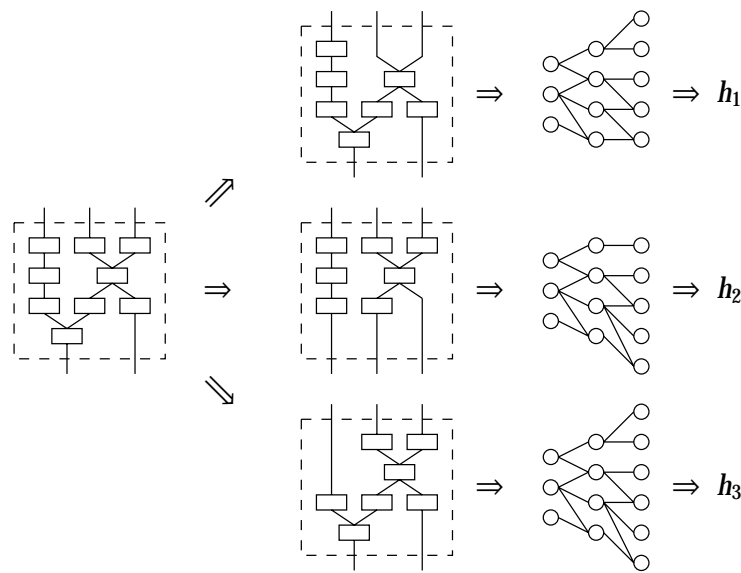
Other systems can be shown to fit the template as well. Examples include well-known systems such as the SPA plan adaptation system by Hanks and Weld (1995), MRL (Koehler, 1994) and the Sherpa replanner of Koenig et al. (2002).

## 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 using that heuristic. 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.

Our approach to unrefinement is sketched in Figure 1. On the lefthand side, we have the current plan  $P$  that is to be unrefined. Then, we compute a number of plans that result from removing actions from  $P$ . For each of the resulting plans, we subsequently use the chosen planning heuristic (for example, a planning graph heuristic is depicted in the figure) 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). In greater detail, this procedure consists of the following steps.

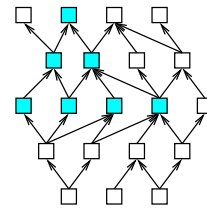
The first step is to decide *which* actions we consider for removal (and thus, for which plans we would like to calculate the heuristic value). Ideally, we would like to consider all possible combinations of actions. However, there are an



**Figure 1: Sketch of the unrefinement heuristic. From the original plan on the left, we derive three subplans and calculate heuristic values ( $h_1$ ,  $h_2$ , and  $h_3$ ) for them using (in this case) a planning graph heuristic.**

exponential number of such combinations to investigate. Since the idea was to quickly calculate an estimate, considering an exponential number of plans is no option. 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 is polynomial in the size of that plan.

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 2: A backward removal tree**

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 trees, until we find a removal tree for which the heuristic shows that it is solvable.

## 6 Experimental Results

The technique to add plan repair capabilities to existing planners, as described in the previous section, has been applied to the VHPOP planner by Younes and Simmons (2003). Here we present initial experimental results with the benchmark set of GPG (Gerevini and Serina, 2000). This set contains problems that feature some aspects of logistics, as we detail below. Each problem set consists of a base problem and a number of derivations (typically 30 or 45). In all cases, the goal is to modify the solution (plan) to the base problem, such that it becomes a solution to the derived problem.

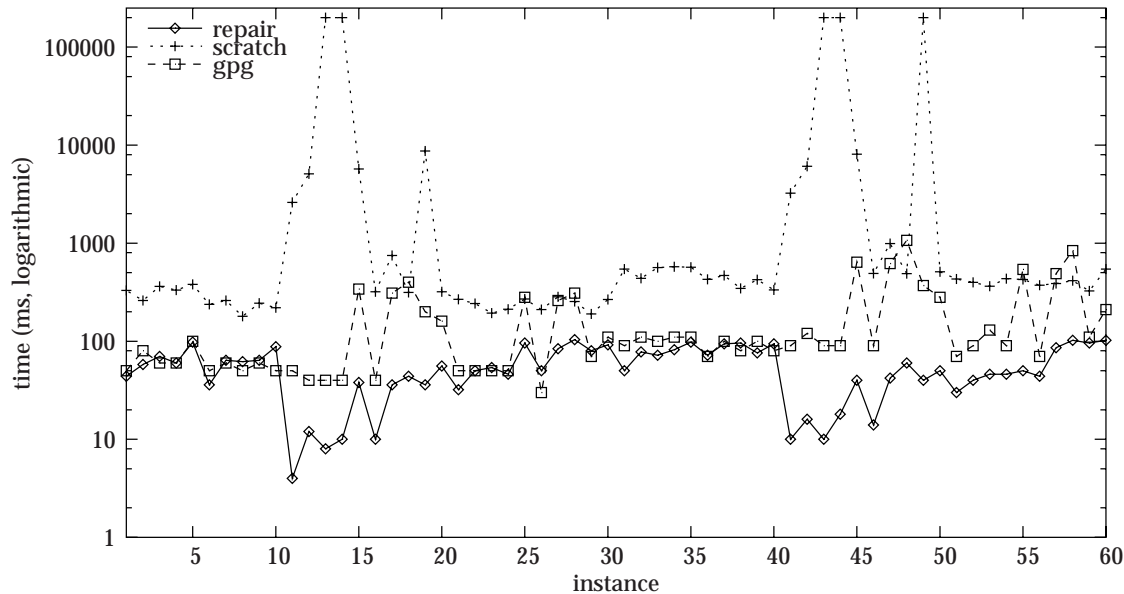
The modification was obtained by using the unrefinement heuristic as described in the previous section. In each instance, we started by adapting the current (base) plan to the new situation: impossible actions (i.e. actions in the plan with preconditions that can never be reached, e.g. when a truck is no longer available) are removed and preconditions of actions that are no longer satisfied by the initial state are marked accordingly. Then, we calculate the cuts of the plan, and use VHPOP to find a solution to the problem thus created.

Figures 4 through 8 show the runtime performance of our system, compared to planning from scratch (using the VHPOP planner) and compared to the GPG plan adaptation system. All times are reported in milliseconds and on a logarithmic scale. For some instances, VHPOP was not able to produce a result within 200 seconds, or ran out of the 500 Mb of memory that was available. In both these cases, the runtime is reported as 200,000 milliseconds. The runtime that is reported for both GPG and our system is the time when the *first* solution was found. Usually, this solution can be optimised further, but here we are interested only in the first solution.

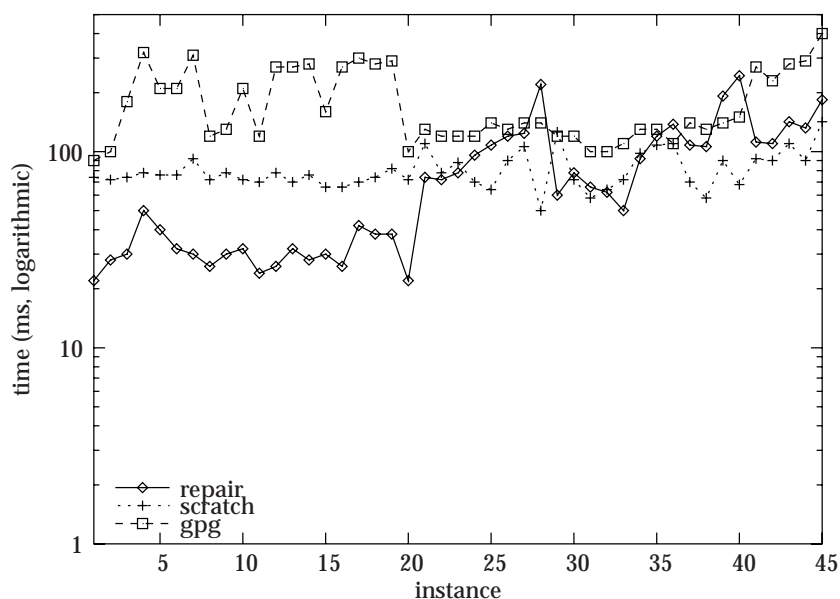
Figure 4 shows the performance on 60 problems in the Gripper domain. Problems in this domain feature a robot, Robby, with two grippers that can move around through a number of rooms. In some of these rooms balls are present, which Robby is to pick up and bring to a specified location. The 60 problems can be divided into two sets: one set in which 10 balls have to be delivered, and a second set in which 12 balls are to be distributed. Both GPG and our system outperform planning from scratch. The difference between both plan adaptation systems is not large, although our system is slightly faster than GPG on most problems.

The next three Figures (5 through 7) present the results from the Logistics domain, with three different base problems. Note that the scale is different here, as all problems could be solved (even from scratch) in under 2 seconds. This domain consists of a number of packages that have to be brought from a source location to their destination. Within a city, goods can be transported by trucks. To transfer goods from one city to another, they have to be brought to the airport from which they can be flown to the airport of the other city. Results on this domain are similar to the results in the previous domain.

The third set consisted of 60 problems in the Rocket domain. The problems in this domain involve transporting goods and people around the world by rockets (which can be thought of as regular airplanes). Several things happen here, as can be seen in Figure 8. First of all, we observe that both GPG and our system are consistently faster than planning from scratch. Secondly, we see that on *average*, both plan repair techniques are equally fast. However, whereas GPG shows a consistent behaviour, our system strongly fluctuates, solving about half

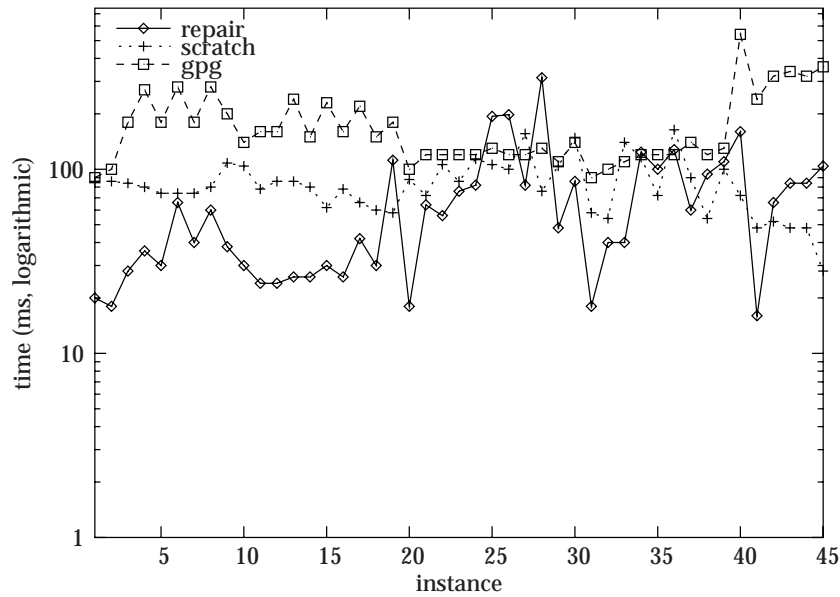


**Figure 4: Runtime for the 60 problems in the Gripper domain. The first 30 problems are with 10 balls, problems 31-60 have 12 balls.**

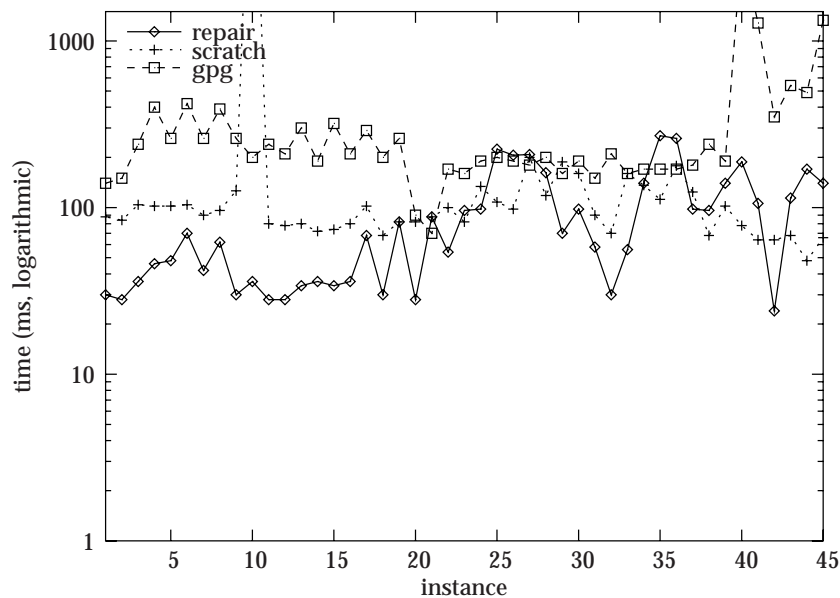


**Figure 5: Runtime for the 45 problems in the Logistics domain (set A)**

of the problems ten times faster than GPG, and half of them ten times slower. An explanation for this behaviour can be found in Figure 9. This figure depicts the size of the search tree that was built during the search. It shows the number of nodes (partial plans) that were generated and the number of nodes that was actually visited during the search. One can see the same strong fluctuations here. In some cases, very few plans are generated and visited. In other cases, the search visits over a thousand plans. The fluctuations are explained by how the goals and initial conditions change with respect to the base problem. For

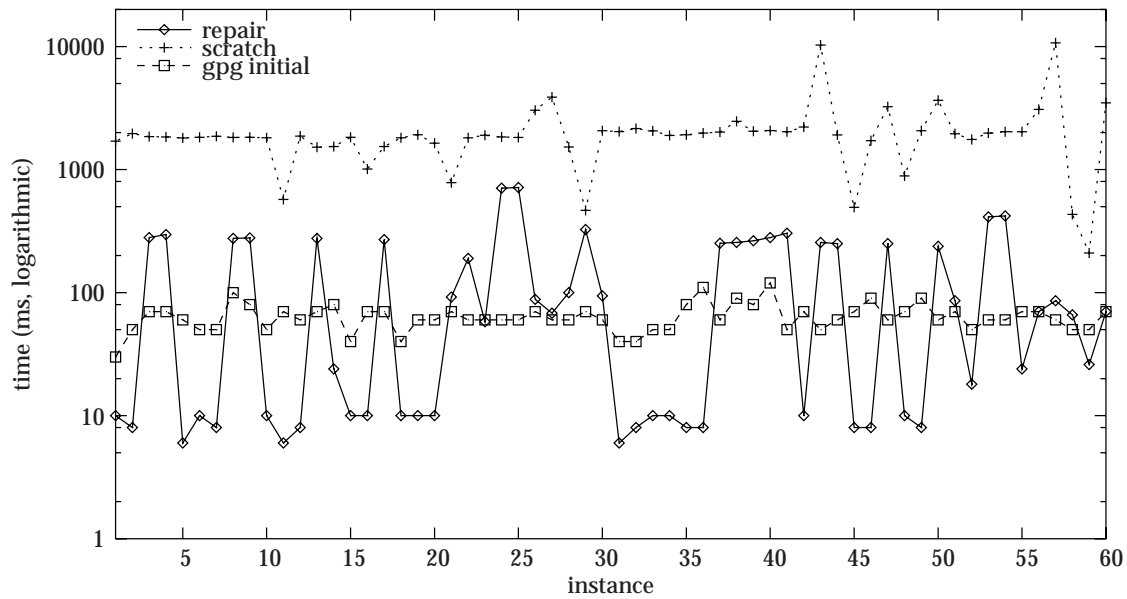


**Figure 6: Runtime for the 45 problems in the Logistics domain (set B)**

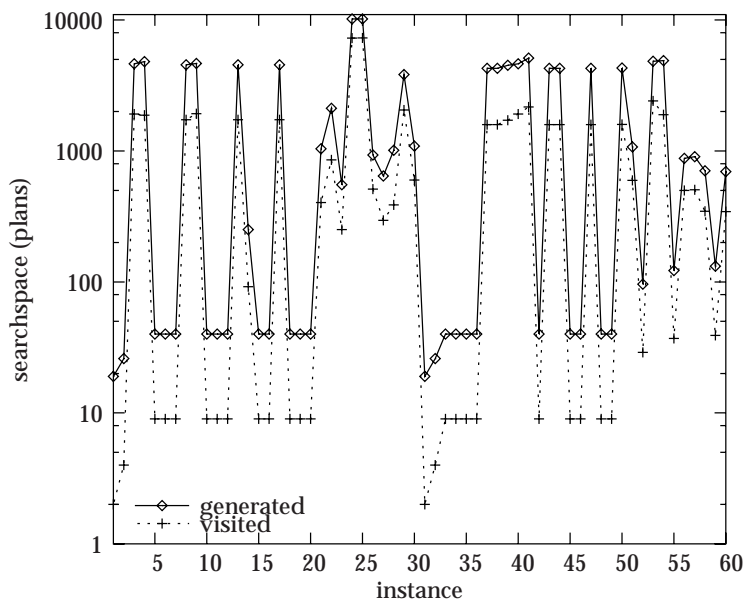


**Figure 7: Runtime for the 45 problems in the Logistics domain (set C)**

some changes, the passenger merely has to board another rocket, for other changes, a rocket has to be flown to the person in question. In the former case, very little work has to be done to repair the plan, whereas in the latter case, some more work is necessary.



**Figure 8: Runtime for the 60 problems in the Rocket domain. The first 30 problems are of set A, the others of set B.**



**Figure 9: Search statistics for the Rocket domain.**

## 7 Discussion

Planning is an important tool in realizing an efficient logistics chain. However, since the domain of transportation problems is highly dynamic, the application of planning in logistics requires efficient techniques for plan adaptation. This paper describes a general framework for plan repair systems, called the *unrefinement planning* approach. This approach elaborates upon Khambampati's (1997) refinement planning theory, which is recognized as a general framework for planning algorithms. The modification consists of including a decision whether to *remove* obstructing actions from the plan, or to *add* additional actions to the plan. The new template clearly shows these two independent, yet related, activities, and allows different strategies to be employed for refining (extending) or unrefining (reducing) a plan. We briefly showed how existing systems (specifically Replan (Boella and Damiano, 2002) and GPG (Gerevini and Serina, 2000), but also other systems) fit in this template.

One of the interesting possibilities given a general framework is that it can be used to devise new plan repair methods. To support this claim, we showed a new unrefinement heuristic. This heuristic is not dependent upon a specific refinement strategy. On the contrary, it is designed to be applicable in conjunction with most existing planning heuristics. This feature was achieved by constructing the heuristic such that the calculation of a value is dependent upon a planning heuristic of choice. To illustrate the strength of this approach, we applied it to the VHPOP planner by Younes and Simmons (2003), turning it into a plan repair system. On a broad range of benchmark problems, this new system is competitive to a recent plan repair system (GPG), as well as planning from scratch. However, for some problems in a particular domain the results are less satisfactory. A possible cause (and thus possibly an improvement) has not been found yet. Besides further improvement of the new system, in future work 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

The authors are part of the Collective Agent Based Systems (CABS) group at the faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology. Roman van der Krogt is supported by the *Freight Transport Automation and Multimodality* (FTAM) research program, carried out within the TRAIL research school for Transport, Infrastructure and Logistics. His project, titled *Incident Management Techniques in Transportation*, is supervised by Cees Witteveen. Mathijs de Weerd is supported by the *Towards Reliable Mobility* (TRM) research program of the Delft University of Technology.

## 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 (LL-NCS 2443)*, pages 183–192. Springer Verlag.
- 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)*, 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.
- 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.
- Koehler, J. (1994). Flexible plan reuse in a formal framework. In *Proc. of the 2nd European Workshop on Planning (EWSP-93)*, pages 171–184. IOS Press, Vadstena, Sweden. ISBN 90-5199-153-3.
- Koenig, S., Likhachev, M., and Furcy, D. (2002). Lifelong planning A\*. Technical Report GIT-COGSCI-2002/2, Georgia Institute of Technology, Atlanta, Georgia.
- van der Krogt, R. and de Weerd, M. (2004). The two faces of plan repair. In *Proceedings of the Sixteenth Belgium-Netherlands Artificial Intelligence Conference (BNAIC-04)*, page to appear.
- 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.
- Younes, H. L. S. and Simmons, R. G. (2003). VHPOP: Versatile heuristic partial order planner. *Journal of AI Research*, 20:405–430.