Anytime Plan-Adaptation for Continuous Planning

Antonio Garrido, Cesar Guzman and Eva Onaindia
Dpto. Sistemas Informaticos y Computacion
Universidad Politecnica de Valencia
Camino de Vera s/n, 46022, Valencia (Spain)
{agarridot, cguzman, onaindia}@dsic.upv.es

Abstract

The execution of a plan in a highly dynamic real-world environment entails facing unexpected events that produce discrepancies between the observed and the predicted state. In a situation like this, we need to fix the flaws, and typically we have two possible options: replan from scratch or repair, i.e. adapt the original plan to the new context. This paper presents an effective method to support the decision making between repairing or replanning. Particularly, we have designed a method that estimates the cost of a *bridge* from the observed state to any reachable goal state in the original plan. We have also endowed this adaptation method with an anytime behaviour to improve the plan quality in terms of both problem metric and stability. The paper also presents some experimental results to evaluate the approach.

Introduction and Motivation

Executing a plan in a real environment entails facing the unexpected changes that happen in the world which manifest as discrepancies between the expected and the observed states. This is frequent in highly dynamic environments involving exogenous events. In this case, it is required not only finding the discrepancy but whether it is relevant to the plan validity or not. Assuming a plan is no longer executable, some fixing is necessary to replace the old plan with a new plan. The two common options to address this problem are plan repair, i.e. adapt the plan to the new situation, or replanning from scratch. Theoretically speaking, both options are equally expensive in the worst case (Nebel and Koehler 1995), but plan repair intuitively seems more efficient in practice since a large part of the plan can be still valid (Gerevini, Saetti, and Serina 2003; van der Krogt and de Weerdt 2005). In some cases, a significant part of the original plan can be entirely reused; or perhaps a certain degree of stability (i.e. maintaining part of the original actions) is required to prevent the executive from executing an entirely new and unknown plan which clearly induces many differences between the original and the new plan (Fox et al. 2006). Hence, plan stability is one of the principal reasons for claiming the preference of plan repair over the alternative of replanning.

The work in (Fox et al. 2006) presents an implementation of a plan repair technique, based on LPG (Gerevini, Saetti, and Serina 2003), which empirically demonstrates that achieves more stability than replanning. Other approaches also regard plan repair as a process of refinement planning (adding actions to the original plan) and unrefinement planning (removing actions from the plan), such as (van der Krogt and de Weerdt 2004). A similar approach to plan adaptation is presented in (Gerevini and Serina 2010), which modifies the original plan within limited temporal windows containing portions of the plan that need to be revised. In all these cases, the term plan stability refers to a measure of the difference, in number of actions, between an original source plan and a new target one, no matter the portions of the plan in which the differences occur.

The aforementioned plan repair techniques are all designed for off-line planning. In continuous planning, i.e. an ongoing and dynamic process in which planning and execution are interleaved, iterative repair techniques seem more appropriate because they support continuous modification and updating of a current working plan in light of changing operating context (Myers 1999; Chien et al. 2000). Then, we can highlight the particular requirements of plan-repair methods for continuous planning:

- Plans must be updated in response to new information and requirements in a timely fashion to ensure that they remain viable. In this sense, it is necessary to promptly take a decision between repair or replanning, taking into account that a severe change in the world could be better solved by generating an entirely new plan.
- Plan-repair based on the analysis of plan dependency structures involves identifying the actions that are no longer executable as a consequence of the plan failure. Obviously, in continuous planning, the interest is not in repairing the whole plan but fixing the earliest portion of the plan as it will be the first to be executed.
- Likewise, minimising the number of changes in the original plan, i.e. maintaining stability, is particularly relevant in the earliest portion of the plan. This is because when a plan is being executed, the executive has likely committed the earliest part of the plan in terms of equipment, resources or time. For instance, minimal perturbation planning is understood in some approaches as a heuristic for respecting the commitments induced by the partial execution of the plan (Cushing and Kambhampati 2005).

This paper presents a plan-fixing method for continuous planning that adapts a failing plan to the new context and contributes with some novel issues:

- A decision-support module, capable of deciding between replanning or repairing in a timely fashion.
- In many situations, it is not only plan stability that matters, but a balanced response between metric (plan cost, makespan, etc.) and stability; our approach adapts the original plan by taking this into consideration.
- Our approach for plan repair puts the emphasis on the first part of the plan (the most urgent to be fixed), so it first aims at keeping the totality of the original plan, and computes a plan that bridges the observed state with the state in which the original plan is applicable. Additionally, we have endowed the repair algorithm with *anytime* capabilities whereby a first solution is rapidly returned, and the solution quality may improve if the algorithm is allowed to run longer.
- We can use any type of PDDL-compliant planner, so our approach is not only domain-independent but planner-independent as well.

Formal Model

We use a planning formalism based on multi-valued state variables, as identified in PDDL3.1[1], rather than binary-valued propositional variables. This formalism is not new, as it is based on the SAS$^+$ planning model (Backstrom and Nebel 1995) and was successfully used in (Helmert 2006). Intuitively, it can be metaphorically considered as a constraint programming formulation for planning (Vidal and Geffner 2006; Garrido, Arangu, and Onaindia 2009), where we model variables with both finite and infinite (real-valued) domains to allow modelling a planning domain more naturally. A fully-instantiated state assigns a unique value to each variable in the problem.

Definition 1 (Multi-valued planning problem) *A multi-valued planning problem is given by the 5-tuple $\mathcal{P} = \langle \mathcal{V}, S_\mathcal{I}, S_\mathcal{G}, \mathcal{O}, \mathcal{M} \rangle$. $\mathcal{V}$ is a set of state variables with an associated finite or infinite domain $\mathcal{D}_v$. $S_\mathcal{I}$ is the initial state over $\mathcal{V}$, where each variable in $\mathcal{V}$ is assigned a given value in $\mathcal{D}_v$. $S_\mathcal{G}$ is a partial variable assignment over $\mathcal{V}$, namely the problem goal state. $\mathcal{O}$ is a finite set of operators $\langle$pre,effs,cost$\rangle$[2] over $\mathcal{V}$ that when applied (i.e., when preconditions hold in a given state) change the values of the variables according to effs —note that no distinction between positive and negative effects is necessary when dealing with a state variable representation. The cost of the operator is considered in $\mathcal{M}$, an expression that encodes the problem metric and that is it used to measure the plan quality.*

[1]See http://ipc.informatik.uni-freiburg.de for more details.

[2]Although we do not consider actions with duration in this paper, our approach is general enough to deal with models of actions that include different types of preconditions/effects such as PDDL2.1, or more expressive ones (Garrido, Arangu, and Onaindia 2009).

Analogously to (Helmert 2006), we define a domain transition graph (DTG) per multi-valued state variable. The idea with this is: i) to describe the behaviour of a variable, in the form of a state transition diagram, thus defining how it changes from one value to another within the planning problem, and ii) to estimate the cost of these transitions and changes, i.e. the paths. Obviously, DTGs only make sense for those variables composed of a finite number of values, that is, with a finite domain. DTGs for variables with infinite domains are not generated.

Definition 2 (Domain transition graph, DTG) *Given a multi-valued variable $v \in \mathcal{V}$, $DTG(v) = (V, E)$ is a digraph, where the vertices V represent the finite domain of values for v and E is a multiset of directed edges $\langle v1, v2 \rangle, v1, v2 \in V(v1 \neq v2)$. Each edge represents a transition between two values, that is, an action application and how it changes the value of the variable. Consequently, transitions are labelled with the preconditions necessary to trigger such an action. Preconditions are encoded as pairs $\langle variable, value \rangle$ standing for the variable and its required value. The associated weight of the transition can be fixed or, more generally, measured in terms of the action metric.*

An additional and helpful structure is the causal graph (Helmert 2006), which maintains the causal dependencies between variables, making it possible establish a *priority* order between them.

Definition 3 (Causal graph and acyclic causal graph) *The causal graph of a multi-valued planning problem, CG($\mathcal{P}$), is a digraph with vertices in $\mathcal{V}$ containing an arc (v, v') iff $v \neq v'$ and the arc is induced by a transition condition, i.e. DTG(v) has a transition with some condition on v'. CG($\mathcal{P}$) is acyclic when: i) there are no cycles in the variables, and ii) no trivially false conditions occur in operators or goals.*

Given a planning problem, the objective is to find a solution plan that, starting from the initial state, satisfies the goal state in a way that optimises the problem metric. Planners return sequential or parallel plans, where actions can co-occur. Our approach subsumes any type of plan by considering a simple, though general, representation. This way, we are interested in a plan as a sequence of states.

Definition 4 (Plan as a linear sequence of states) *A plan is defined by the chronologically ordered sequence of states $\Pi = \{S_0, S_1, \ldots S_n\}$, where each S_i is a fully-instantiated state that results from the effects of actions (see Figure 1). S_0 is the initial state and $S_\mathcal{G} \subseteq S_n$, which means that the goal state $S_\mathcal{G}$ is satisfied in the last state of the plan S_n.*

Note that the same fully-instantiated state can satisfy different goal states and vice versa, i.e. the same goal state can be satisfied in different states. This is particularly relevant in our repairing approach, and allows us to reuse part of a plan when calculated in a regressive way.

Definition 5 (Regressed goal state) *A regressed goal state is a goal state, i.e. a partial variable assignment over $\mathcal{V}$, that is calculated regressively. Given a plan $\Pi = \{S_0, S_1, \ldots S_n\}$, a regressed goal state is calculated for*

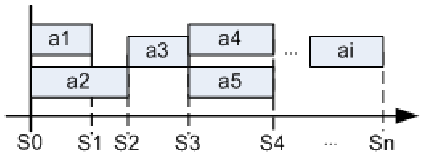

Figure 1: Plan as a sequence of states. This representation is valid for sequential, parallel and even temporal plans

each S_i starting from the last state S_n and regressing state by state all over Π. A regressed goal state comprises the minimal set of preconditions necessary to execute the next part of the plan. In other words, if the regressed goal state for state S_i holds in such a state, the remaining plan from that state (actions in $]S_i..S_n]$) can be executed as is.

Regressed goal states are inspired by the mechanism used in triangle tables defined in (Fikes, Hart, and Nilsson 1972) and calculated in Algorithm 1, which is presented below.

It is important to note that the number of regressed goal states and, more generally, states in a plan does not correspond exactly with the number of actions, but with the number of state changes that actually happen in the plan. As can be seen in Figure 1, this not only depends on the number of actions in the plan, but also on the type of plan, parallel or sequential, and the model of actions. First, if the effects of two or more concurrent actions occur at the same time, this means one single state change. Second, the model of actions defines different levels of expressiveness. In a classical model, effects only happen at the end of the action, so the states only change at that point. However, in other (temporal) models, effects may happen when the action starts or ends, so the state can change twice. All in all, defining a plan as a sequence of states provides us with a flexible representation to include any type of plan and action formalism.

Anytime Plan-Adaptation Approach

The underlying idea of our overall proposal is to execute and monitor a plan $\mathcal{P} = \langle \mathcal{V}, S_\mathcal{I}, S_\mathcal{G}, \mathcal{O}, \mathcal{M} \rangle$. If any discrepancy between the expected and observed context appears during execution[3], i.e. reaching an unexpected state because exogenous events change some *static* information in the world, or when an action fails to execute or achieve part of its effects, it becomes necessary to come up with a fixing mechanism via repairing or replanning. So, the first issue is to decide whether plan-repair is more convenient than replanning; and we need this decision to be simple, effective and timely fashion. The basic idea for repairing is to build a bridge, to be used as a prefix of the original plan, that connects the observed (real) state to one of the states of the original plan

[3]Although it is also possible to have discrepancies in the problem goals, i.e. when new goals appear and others disappear, we are using an on-line execution+monitoring+fixing approach. This means that we focus on the discrepancies between the expected state and the real one, which may entail solving preconditions for actions that need to be executed immediately or in the short-term, but not in the end of the plan.

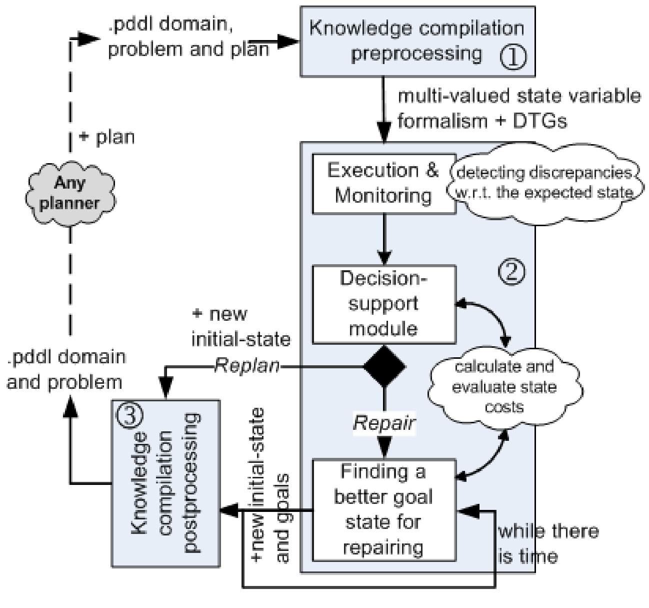
.pddl domain, problem and plan
Knowledge compilation preprocessing ①
multi-valued state variable formalism + DTGs
+ plan
Any planner
Execution & Monitoring
detecting discrepancies w.r.t. the expected state
②
Decision-support module
+ new initial-state
Replan
.pddl domain and problem
calculate and evaluate state costs
Repair
③ Knowledge compilation postprocessing
+new initial-state and goals
Finding a better goal state for repairing
while there is time

Figure 2: The three stages of our approach

in order to reuse its remaining actions. Loosely speaking, we transform the information about the observed state and the connection state of the original plan as the desired initial state and goals, respectively, into a new planning problem $\mathcal{P}'$ that can be solved by any planner, and does not require to implement an ad-hoc adapting module or special flaw-fixing technique. This way, we convert this fixing problem into a new, and probably much smaller and simpler, planning problem. On the other hand, if the decision-support module decides to replan we simply substitute the initial state of the original planning problem for the observed state and solve it again. The schema for our plan fixing approach consists of three stages, as depicted in Figure 2, and described next.

1. Preprocessing Stage. Creating the Multi-valued Planning Problem and Structures

As planning domains are usually encoded in a PDDL standard format without a clear definition of variables[4], we use a preprocessing phase to translate this encoding into a multi-valued state variables formalism. The input is a classical PDDL domain+problem, whereas the output is a collection of files that represent a multi-valued planning problem as specified in Definition 1. As a second preprocessing phase, we have a knowledge compilation module that builds one DTG per each of the multi-valued state variable of the problem, and compiles the ways in which it may change its value through action application. In this stage we use the

[4]In PDDL it is common to have a huge number of binary propositions such as `(at truck1 city1)`, `(at truck1 city2)`,...`(at truck1 city`$_n$`)` with a $\{$`true, false`$\}$ domain, rather than a general variable that subsumes all these propositions, such as `(location truck1)` with a $\{$`city1, city2,...city`$_n$$\}$ domain.

`translate` and `preprocess` tools of LAMA[5].

2. Plan Fixing Stage

Once the plan is executed and a discrepancy is found by the monitoring module, the plan fixing stage really begins. Intuitively we expect to reuse an important amount of the original plan, thus minimising the adaptation process. But in some cases replanning may show more reasonable, particularly in those cases where the discrepancies are significant and the plan has been *seriously damaged*. In other words, we do not want to achieve stability *at any price*, which may lead us to a plan significantly worse in terms of the problem metric, but to a balanced solution. Consequently, we first need to assess whether repair seems to be more promising than replanning from scratch.

Decision-Support Module. In this module, we start with the translation of the remaining, unexecuted original plan Π into a linear sequence of states as meant in Definition 4. More formally, $\Pi = \{S_i, \dots S_j, \dots S_n\}$ with S_i equals to the expected state and $S_{\mathcal{G}} \subseteq S_n$. Thus, from now on we work with an enumeration of states rather than on a more complex structure of actions in a sequential or parallel order.

We successively generate regressed goal states as shown in Algorithm 1. As indicated in Definition 5, holding a regressing goal state means that the remaining plan is executable. This is very important, as it allows us to focus on the satisfaction of a regressed goal state and ignore the subsequent part of the plan. In particular, if we focus on satisfying the last goal state, GS_n, from the current observed state we are actually doing replanning (nothing from the original plan is reused). On the contrary, if we focus on satisfying the first goal state, GS_i, we are reusing the entire original plan and we are in the most extreme situation for repairing; we try to create a bridge from the observed state to the goal state GS_i and resume the original plan. Clearly this does not always lead to the best solution but, intuitively, it leads to the most stable, i.e. conservative, adaptation approach. Considering this, if we estimate and evaluate the cost of a plan for each of these two situations we will be able to opt for a fixing mechanism oriented to either replanning or repairing.

For simplicity matters, each plan is splitted into two subplans to be concatenated: i) a head, $\{S_i..S_j\}$; and ii) a tail, $\{S_j..S_n\}$. Head is created with the actions that are executed from S_i to the intermediate state S_j and tail with the actions executed from S_j to S_n. Let us assume that $S'_{\mathcal{I}}$ is the current observed state (the new initial state) that diverges from the predicted state. The decision-support module estimates two new plans to solve these discrepancies on the basis of the original plan Π. The former, Π_{replan}, is calculated from $S'_{\mathcal{I}}$ to GS_n. Obviously, tail(Π_{replan})=$\emptyset$, because there are no actions in the plan beyond S_n, and head(Π_{replan}) is unknown and needs to be calculated. The latter, Π_{repair}, is calculated from $S'_{\mathcal{I}}$ to GS_i; tail(Π_{repair})=Π, as the whole remaining plan is reused, thus maximising the plan stability, whereas head(Π_{repair}) is also unknown and needs to be calculated. In both cases, head needs to be calculated before evaluating which plan is the best one. One plan is the best when its cost(head)+cost(tail) is the best. The cost is calculated according to the problem metric As usually done in heuristics, we assess the plan cost in terms of a real cost (for tail) and an estimate (for head). More particularly, $Cost(\Pi_{replan}) = Cost(S'_{\mathcal{I}}..GS_n) + 0$ and $Cost(\Pi_{repair}) = Cost(S'_{\mathcal{I}}..GS_i) + Cost(tail(\Pi_{repair})$.

```
1: Input: $\Pi = \{S_i, \dots S_j, \dots S_n\}$, its actions and $S_{\mathcal{G}}$
2: Output: the sequence of (regressed) goal states for $\Pi$, $\{GS_i, \dots GS_j, \dots GS_n\}$
3: $GS_n \leftarrow S_{\mathcal{G}}$ //note that $S_i$ represents the expected state
4: $GS_{\mathtt{r}} \leftarrow \emptyset, \forall \mathtt{r} : i..n-1$
5: $\mathtt{r} \leftarrow n$ //init the regression counter
6: while $\mathtt{r} > i$ do
7:    for all goal $\mathtt{g} \in GS_{\mathtt{r}}$ do
8:       if $\exists$ action $\mathtt{a}$ that supports $\mathtt{g}$ in $GS_{\mathtt{r}}$ then
9:          $\mathtt{t} \leftarrow$ time when $\mathtt{a}$ starts
10:         $GS_{\mathtt{t}} \leftarrow GS_{\mathtt{t}} \cup pre(\mathtt{a})$
11:      else
12:         $GS_{\mathtt{r-1}} \leftarrow GS_{\mathtt{r-1}} \cup \{\mathtt{g}\}$
13:   $\mathtt{r} \leftarrow \mathtt{r} - 1$
```

Algorithm 1: Generating the goal states for a given plan

The function *Cost*(*init_state..goal_state*) is calculated applying Dijkstra's algorithm, and in our case it represents the cost between an initial and a goal state. It simply consists in finding and storing the best-path, as a sequence of actions that will form the plan, between two nodes (values of a variable) in a DTG. We use Dijkstra's algorithm not only to estimate the cost, but also to estimate a plan for the head. More particularly, given an initial state $i\_s(\mathcal{V}) = \{\langle var1 = init\_val1\rangle, \langle var2 = init\_val2\rangle, \dots \langle var_n = init\_val_n\rangle\}$ and a goal state as a partial variable assignment $g\_s(\mathcal{V}) = \{\langle var1 = goal\_val1\rangle, \langle var2 = goal\_val2\rangle, \dots \langle var_i = goal\_val_i\rangle\}$, we define:

$$Cost(i\_s, g\_s) = \sum_{\forall var_i \in g\_s(\mathcal{V})} best\_dist(init\_val_i, goal\_val_i),$$

where *best_dist* is calculated in each DTG(var$_i$). This is a heuristic estimate because the cost is calculated for each variable in an independent way; that is, it does not take into account positive or negative interactions among variables and the actions application. It is, therefore, possible to have an action that changes two different variables at the same time, which in some cases it means to have it counted twice (over-estimate) but in others it would require more actions to reassign the original value of the second variable (under-estimate). Also the variables are processed according to their causal dependency order, as defined in the acyclic causal graph of Definition 3. In particular, variables with no predecessors are selected first, and so on. For example, if a package can be moved by means of a truck, the package's location variable depends on the truck's location variable and the latter is studied first.

On the other hand, *Cost*(*sub-plan*) returns the real cost of the actions in the sub-plan in terms of the prob-

[5] We thank S. Richter, M. Westphal and M. Helmert for providing us with tools, as separated programs, of their LAMA planning system (Helmert 2006; Richter and M. 2008).

lem metric. $Cost(tail(\Pi_{replan}))$ is always zero and $Cost(tail(\Pi_{repair}))$ is calculated from the actions in tail.

Given the two estimated plans Π_{replan} and Π_{repair}, the decision-support module applies Algorithm 2 to choose the most adequate one. The first criterion is based on the metric cost and it can be conclusive if the difference in the cost is big enough (in our experiments we use $\alpha = 0.05$). Otherwise, we use a second criterion based on stability that checks each plan with the original one and selects the one that maximises the stability. Lower values of α make the function more metric-oriented, whereas higher values make it more stability-concerned.

```
1: Input: $\Pi_{replan}$ and $\Pi_{repair}$
2: Output: the plan to be used for fixing the discrepancies
3: if $|Cost(\Pi_{replan}) - Cost(\Pi_{repair})| \leq \alpha * Cost(\Pi_{repair})$ then
4:     return arg. $\max(Stab(\Pi_{replan}), Stab(\Pi_{repair}))$
5: else
6:     return arg. $\min(Cost(\Pi_{replan}), Cost(\Pi_{repair}))$
```

Algorithm 2: Decision-support function: replanning *vs.* repair

The function $Stab(\Pi_r)$ is calculated as a relative percentage of stability in comparison to the original plan Π as:

$$Stab(\Pi_r) = \frac{\text{number actions of } \Pi_r \in \Pi}{\text{number actions of } \Pi_r}$$

In many cases the actions in Π_{replan} also appeared in Π, but the numerator of this fraction is usually higher for Π_{repair} than for Π_{replan} —note that at least, tail(Π_{repair})=Π. If $Stab(\Pi_{replan}) = Stab(\Pi_{repair})$ we return the plan with the max number of actions in Π.

If the decision-support module opts for replanning, the flow continues with the third stage and with a planner to really solve the replanning problem (see the entire cycle in Figure 2). If the decision is to repair we also continue with the third stage to solve the repairing problem, but if there is available time an anytime module tries to find a better goal state that could lead to a better Π'_{repair} and, eventually, to a better solution.

Finding a Better Goal State for Repairing. An Anytime Approach. If the decision-support module has decided that repairing seems the most reasonable option, it is possible that the first goal state, GS_i in Π, is not the most adequate one to do the repairing operation. In other words, given a sequence of goal states $\{GS_i, \ldots GS_j, \ldots GS_n\}$ generated by Algorithm 1, the easiest way to reuse/adapt the original plan is to create a bridge to GS_i, which actually entails the biggest tail. But perhaps this is not the most efficient way, and a goal state GS_j could lead to a better solution —undoubtedly it would reuse fewer actions in tail but the head may be of better quality. Clearly, the evaluation of all the goal states is prohibitive for a decision-support module that must provide an efficient answer in a real-execution environment, but in those cases where there is available time an anytime approach can find a better way of adapting the plan. And the more time we have, the better the solution will likely to be.

The way to deal with this anytime approach is a simple generalisation of the idea used in the decision-support module to evaluate goal states. Instead of using simply GS_i, we now progressively analyse GS_{i+1}, GS_{i+2} and so on, and check whether a goal state is heuristically better than GS_i. We start from GS_{i+1} because we expect to reuse the original plan as much as possible, that is, we are again highly stability-concerned. Actually, we do not expect to explore too many levels, and rarely reach levels close to GS_n as they would be nearly like doing replanning. This process proceeds as indicated in Algorithm 3. The most interesting issue in the algorithm is the use of a block-size in steps 6–7 that focuses on the best $\Pi_{repair}(GS_j)$ —plan repair that is generated for the goal state GS_j—, thus reducing the number of calls to the third stage in step 9. But in our experiments block-size=1 provides good enough results without a significant overhead.

```
1: Input: $best\_\Pi_{repair}$, block-size, and index of the last $GS$ explored
2: Output: $\Pi_{repair}$ that improves $best\_\Pi_{repair}$
3: $best\_\Pi_{rep} \leftarrow best\_\Pi_{repair}$
4: r $\leftarrow$ index of the last $GS$ explored
5: while $\exists$ available time do
6:     $best\_\Pi_{rep} \leftarrow$ arg. $\min(Cost(\Pi_{repair}(GS_j)), Cost(best\_\Pi_{rep})), \forall j \in [\mathtt{r}, \mathtt{r}$+block-size]
7:     r $\leftarrow$ r+block-size
8:     if $best\_\Pi_{rep}$ improves $best\_\Pi_{repair}$ then
9:         return $best\_\Pi_{rep}$ //to invoke the third stage
```

Algorithm 3: Anytime approach to improve the repairing process

3. Translation Stage. Generating the New Planning Problem

After the second stage has decided whether replan or repair, the third stage compiles all the necessary information to generate a new planning problem to be solved by a given planner. Let us assume that the original planning problem is $\mathcal{P} = \langle \mathcal{V}, S_{\mathcal{I}}, S_{\mathcal{G}}, \mathcal{O}, \mathcal{M} \rangle$. The problem to be generated in this stage replaces the original initial state by a new initial state $S'_{\mathcal{I}}$ with the observed state. The problem goals are different depending on the option to be done. If we are doing replanning the goals do not change, i.e. they are $GS_n = S_{\mathcal{G}}$. If we are doing repairing the goals represent the best goal state, GS_1 in the first iteration but others GS_j if we perform the anytime approach explained in Algorithm 3. Typically, these new goals are usually simpler than $S_{\mathcal{G}}$. Note that the original goals in $S_{\mathcal{G}}$ will be really achieved after concatenating the new plan and the calculated tail. In any case, the variables, operators and problem metric remain the same.

Experimental Results

A complete evaluation of our approach would entail a real on-line scenario for execution+monitoring+fixing but, in order to check its validity in a large number of situations, we use four domains of the last IPCs (`driverlog`, `elevators`, `logistics` and `rovers`) to simulate this. In our experiments, input plans were generated by LPG, but we use LAMA as the planner to solve the problems generated in the third stage. As pointed in (Fox et al. 2006), we use different planners to avoid interactions that may appear by using the same planning strategy when creating the original plan and fixing the flaws. We have also tested input plans given by other planners (see below), but they are not shown here due to the lack of space. Flaws are generated by randomly executing up to five (noisy) actions of the domain, similarly to (Gerevini and Serina 2010), which produce discrepancies *w.r.t.* the expected state. In order to evaluate the anytime approach, we set a 20 s limit in which our approach makes a decision between replanning or repairing, returns the first solution and improves it, if possible. We use plan cost as the metric to be minimised, being action costs randomly generated. All tests were run on a Debian Linux computer with a 2.40GHz Intel Core Duo and 3Gb of RAM.

Figure 3 shows the quality of our solutions in terms of problem metric and stability. We depict the solution found by the replanning option in order to compare it with the first and best solution provided by our approach. As can be seen, the metric of the first solution is usually better than replanning, although in a few cases the replanning option also returns good results. Clearly, the best solution is better than the first solution, but in many cases (elevators and logistics) the quality cannot be improved within the given time limit, which gives us an overall idea of the good quality of the first solution. The differences are more remarkable *w.r.t.* the stability, where the first solution always returns a highly stable plan. On the contrary, the best solution is usually less stable than the first solution (sometimes finding a better quality plan has a negative impact on stability), but in 94% of problems it is still better than replanning.

Figure 4 shows the results for the runtime. The differences here are minimal, and finding the first solution is not usually much faster than replanning, though these differences increase in bigger problems. To date our interest has not focussed on code optimisation and we are aware of an important bottleneck in the implementation of the cost functions used in the second stage. After dealing with this issue, we expect to enhance the results in the runtime. To sum up, Table 1 gives us an idea about the average times spent in our approach to take a decision and find the solution within the time limit. Again, these values are very reasonable, but in the driverlog domain they are worse because the time taken by the last problems have an adverse effect on the average.

Finally, although not shown in these experiments for lack of space, if we use input plans provided by other planners the results are very similar: our approach is significantly better than replanning in terms of problem metric and stability, but not as good in the runtime. However, these differences are less significant when we use the same planner for the input plan and for fixing the flaws. In particular, if we use LAMA in both cases the differences between replanning and repairing are scarce, which points to a more difficult way to improve the quality of the plan by using the same planning strategy. Although our approach is open to different planners after the third stage, we still have some problems in the generation of the new PDDL codification. The reason for this is that the necessary static information encoded in PDDL problems is missing because it has been removed in our preprocessing stage. We are now working to overcome this subtle limitation in our implementation and thus testing our approach in a wider range of planners.

Domain	Dec-support	1st soln.	Best soln.
driverlog	2.71 (15/4)	2.85	3.52
elevators	0.37 (18/2)	0.48	0.48
logistics	0.49 (16/4)	0.61	0.63
rovers	0.91 (16/4)	1.01	1.08

Table 1: Summary results. Average running times (per domain) taken by the decision-support, the first and the best solution found in the 20 s limit. Values *x/y* represent the number of times that the decision-support opts for repairing/replanning in each domain

Discussion and Conclusion

The main strength of our fixing approach is twofold. First, it provides a simple but effective decision-support module to choose between replanning or repairing the plan. This method is quite flexible and takes into consideration both metric and stability criteria. It is also very useful because in some cases it opts for replanning from scratch, as it seems to lead to a better quality plan than repairing. Second, when repairing is the selected option, rather than developing new methods and techniques for flaw-repairing, which are usually solver-dependent, we convert the new problem into a simpler (and smaller) one that creates a bridge between the observed state and the last portion of the remaining plan. Hence, we tackle plan-adaptation as an extension of planning. Essentially, we create a problem that can be solved by any planner, thus being solver-independent, and usually involves a plan containing just a few actions. After this, the whole final part of the plan —literally, the plan tail— is reused. Additionally, the anytime approach allows us to search other options for repairing by finding better goal states to improve the quality of the solution.

Up to now, our approach addresses discrepancies only in the current state of execution, but discrepancies may also arise in the problem goals and actions with preconditions that are no longer reachable. If the discrepancies in the problem goals mean fewer goals, our approach can be used as is. But, if the domain needs to be changed or new problem goals are required our approach may not be valid, which is a limitation. The idea of reusing a portion of an original plan as the tail of a new one may turn out invalid if a reused action affects negatively to one of the new goals. Although our approach cannot be directly used here, the underlying idea remains valid. Instead of splitting our plan only into two sub-plans (head+tail), and find one regressed goal state, we need to work with more goal states and split the plan into more fragments. This is part of our current line of research.

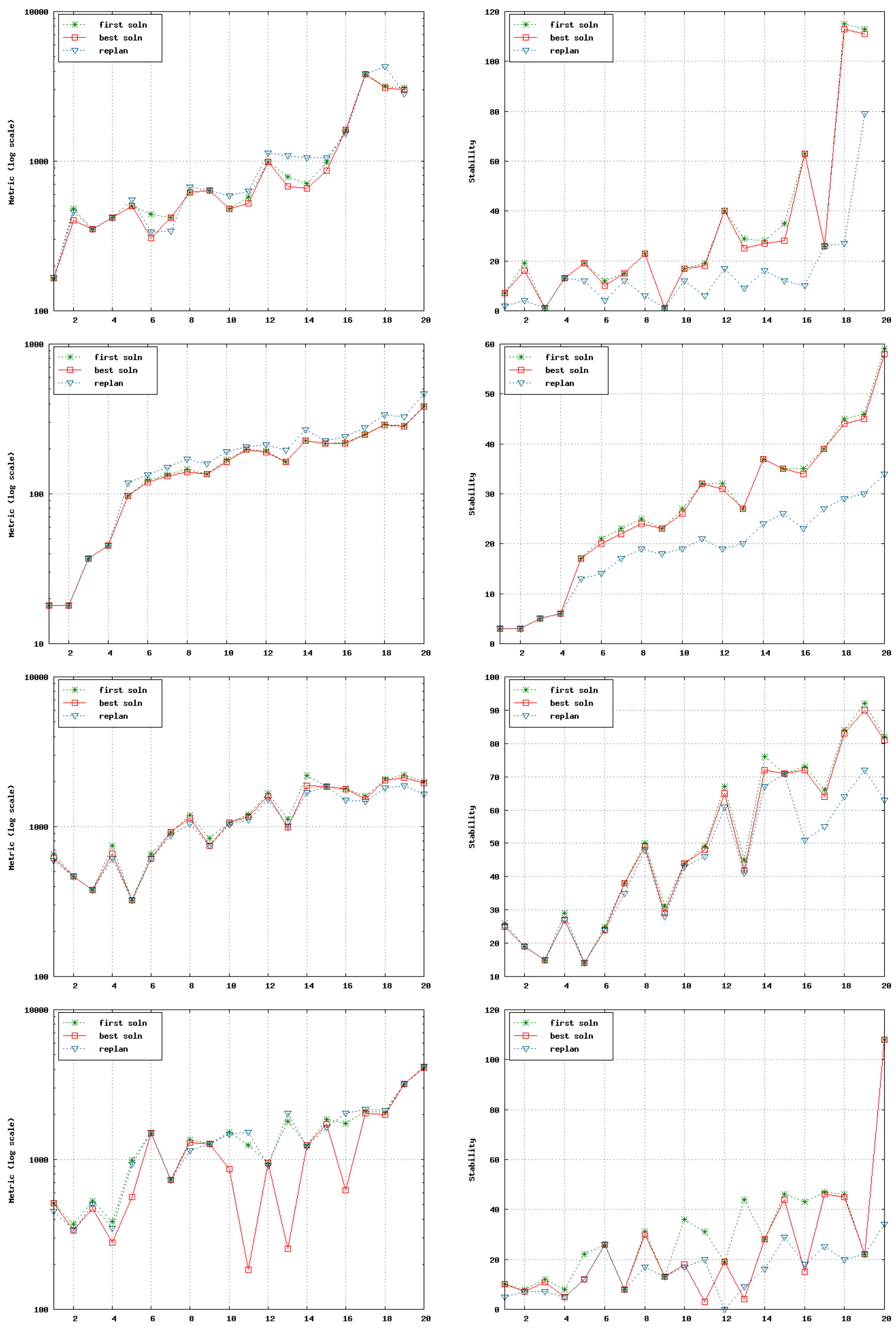

Figure 3: Problem metric (1st column) and stability (2nd column) results for the `driverlog`, `elevators`, `logistics` and `rovers` domains, respectively

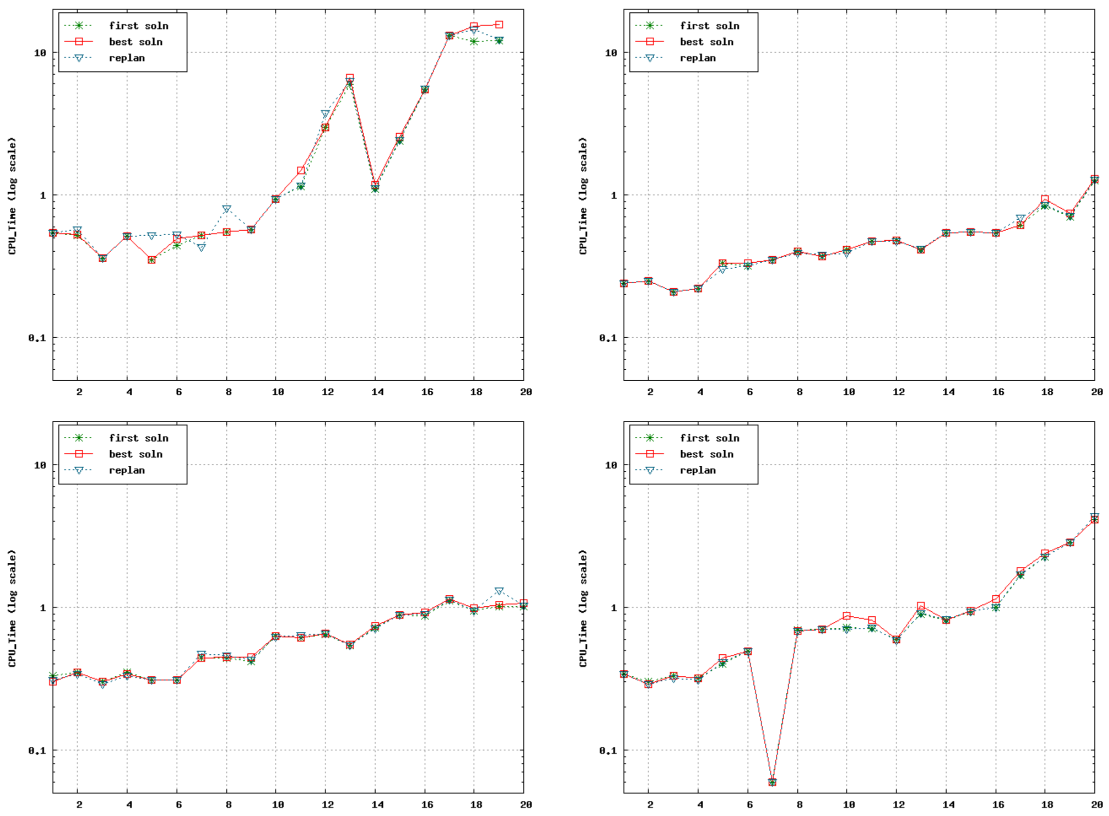

Figure 4: Runtime results for the `driverlog`, `elevators`, `logistics` and `rovers` domains, respectively

Acknowledgments

This work has been partly supported by the Spanish MICINN under projects TIN2008-06701-C03-01 and Consolider Ingenio 2010 CSD2007-00022, the Valencian Prometeo project 2008/051 and Vicerrectorado de Investigacion de la UPV by PAID-05-09 4365-20100087.

References

Backstrom, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11(4):625–655.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *AIPS*, 300–307.

Cushing, W., and Kambhampati., S. 2005. Replanning: a new perspective. In *Poster Programme ICAPS-2005*.

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4):349–371.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In *ICAPS-2006*, 212–221. AAAI Press.

Garrido, A.; Arangu, M.; and Onaindia, E. 2009. A constraint programming formulation for planning: from plan scheduling to plan generation. *JOSH* 12(3):227–256.

Gerevini, A., and Serina, I. 2010. Efficient plan adaptation through replanning windows and heuristic goals. *Journal of Algorithms in Cognition, Informatics and Logic, to appear.*

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs in LPG. *JAIR* 20:239–290.

Helmert, M. 2006. The fast downward planning system. *JAIR* 26:191–246.

Myers, K. 1999. CPEF: a continuous planning and execution framework. *AI Magazine* 20(4):63–69.

Nebel, B., and Koehler, J. 1995. Plan reuse versus plan generation: a complexity-theoretic perspective. *Artificial Intelligence* 76:427–454.

Richter, S., and M., W. 2008. The LAMA planner. using landmark counting in heuristic search. In *IPC@ICAPS-2008*.

van der Krogt, R., and de Weerdt, M. 2004. The two faces of plan repair. In *Proc. Belgium-Netherlands Conference on Artificial Intelligence (BNAIC-04)*, 147–154.

van der Krogt, R., and de Weerdt, M. 2005. Plan repair as an extension of planning. In *ICAPS-2005*, 161–170.

Vidal, V., and Geffner, H. 2006. Branching and pruning: an optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170:298–335.