SimPlanner: An Execution-Monitoring System for Replanning in Dynamic Worlds

Eva Onaindia, Oscar Sapena, Laura Sebastia, and Eliseo Marzal

Dpto. Sistemas Informaticos y Computacion
Universidad Politecnica de Valencia
{onaindia,osapena,lstarin,emarzal}@dsic.upv.es

Abstract. In this paper we present SimPlanner, an integrated planning and execution-monitoring system. SimPlanner allows the user to monitor the execution of a plan, interrupt this monitoring process to introduce new information from the world and repair the plan to get it adapted to the new situation.

1 Introduction

Research on AI planning usually works under the asumption that the world is accessible, static and deterministic. However, in dynamic domains things do not always proceed as planned. Interleaving planning and executing brings many benefits as to be able to start the plan execution before it is completed or to incorporate information from the external world into the planner [8]. Recent works on this field analyse the combination of an execution system with techniques as plan synthesis or anticipated planning [3]. Other works on reactive planning [10] are more concerned with the design of planning arquitectures rather than exploiting the capabilities of the replanning process [9].

SimPlanner is a planning simulator that allows the user to monitor the execution of a plan and introduce/delete information at any time during execution to emulate an external event. It is a domain-independent, synchronous replanner that, like other planning systems [10], avoids generating a complete plan each time by retaining as much of the original plan as possible. SimPlanner has been specially designed for replanning in STRIPS-like domains and has been successfully applied to a great variety of different domains. Additionally, the replanner obtains the optimal solution with respect to the number of actions for most of the test cases.

2 Monitoring the Execution of a Plan

Replanning is introduced during plan execution when an unexpected event occurs in the world. One problem with unexpected effects is deciding how they interact with the effects of the action that was currently being executed. Our solution is to assume the action took place as expected and simply to insert the unexpected effects after the execution of the action.

P. Brazdil and A. Jorge (Eds.): EPIA 2001, LNAI 2258, pp. 393–400, 2001.

When an event is produced it is necessary to verify the overall plan is still executable. Many replanning systems only perform a precondition checking to verify whether next action is executable. This option is less costly and much easier to implement in many real applications where the sensory system does not capture all predicates that have changed in the problem. However, this apparently efficient approach may turn out to be inefficient in the long term as many unnecessary actions might be introduced due to changes in the plan are not foreseen enough time in advance.

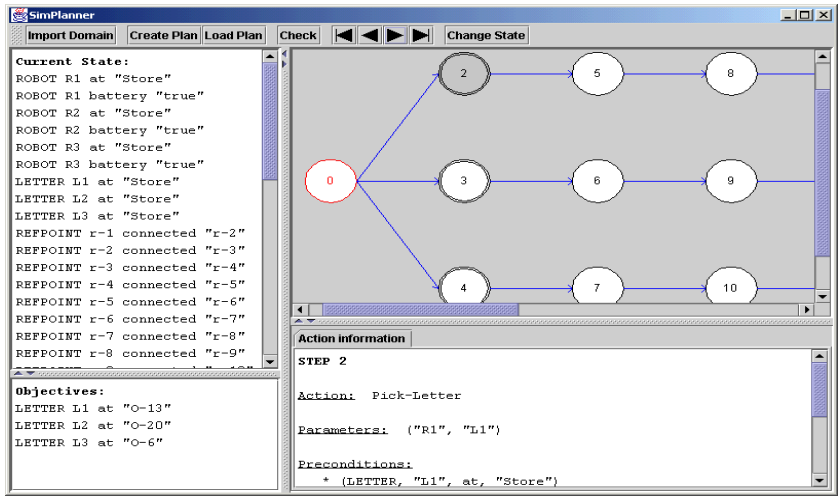

Fig. 1. SimPlanner main interface window

Figure 1 shows the graphical interface to monitor a plan execution. The problem corresponds to one of the instances in the robot domain which SimPlanner has been tested on. In the left upper part of the screen it is shown the literals of the current state of the execution. On the right side of the screen a graph representing the plan under execution is displayed. The circles stand for the actions in the plan. Those actions ready to be executed at the next time step are double-circled.

3 Replanning During Execution

The replanning algorithm starts from the current state in the plan execution, when the unexpected event has been input. The objective is to discover which state should be reached next in the problem so as to retain as much of the original plan as is reasonable without compromising the optimal solution. The following two sections explain the algorithm in detail and provide an example to clarify SimPlanner behaviour.

3.1 SimPlanner Algorithm

Initially, the user is monitoring the execution of a plan $P = (a_0 \rightarrow \cdots \rightarrow a_n)$ when an expected event is introduced in the system. The remaining plan to be executed is defined as $\Pi = (a_0\prime \rightarrow \cdots \rightarrow a_n)/\forall a_i \in \Pi \rightarrow a_i \in P$ and a_i has not been executed yet. $a_0\prime$ represents the new current situation (an initial action with effects and no preconditions).

Algorithm SimPlanner $(\Pi) \rightarrow$ plan Π'

---

1. Build a Problem Graph **(PG)** alternating literal levels and action levels $(L_0, A_1, L_1, A_2, ...)$, where:

A_j=$\{a_j$: $\mathsf{Pre}(a_j) \in L_{j-1} \wedge a_j \notin A_i, i < j\}$
$L_j = L_{j-1} \cup \mathsf{Add}(a)\ \forall a \in A_j$

2. Compute the necessary state, $S(a)$, for each $a \in \Pi$

$$S(a_i) = \begin{cases} \{l : l \in \mathsf{Pre}(a_n)\}, i = n \\ \bigcup_{a_j \in \mathsf{Next}(a_i)} S(a_j) \bigcup (\mathsf{Pre}(a_i) - \mathsf{Add}(a_i)) - \bigcup_{a_k \in \mathsf{Paral}(a_i)} \mathsf{Add}(a_k), \text{ otherwise} \end{cases}$$

where:

$\mathsf{Next}(a_i) = \{a_j \in \Pi \wedge a_i \rightarrow a_j\}$
$\mathsf{Succ}(a_i) = \bigcup_j a_j \in \Pi / a_i \rightarrow \cdots \rightarrow a_j$
$\mathsf{Paral}(a_i) = \{a_j \in \Pi \wedge a_j \notin \mathsf{Succ}(a_i) \wedge a_i \notin \mathsf{Succ}(a_j)\}$

3. Compute the set of possible reachable states $RS = \{S(c_1), \ldots, S(c_n)\}$, where:

$c_i = \{a_1, a_2, \ldots\}/\forall p \in \mathsf{Path}(\Pi) \rightarrow \exists!\ a_k \in p \wedge a_k \in c_i$

$\mathsf{Path}(\Pi)$ the set of all possible paths between $a_0\prime$ and a_n, and $S(c_i) = \bigcup_{\forall a_j \in c_i} S(a_j)$.

4. Select the optimal reachable state $S_{opt} = argmin\{f(x) : x \in RS\}$, where $f(x) = g(x) + h(x)$.

5. Construction of the final plan $\Pi' = (a_0\prime \rightarrow \cdots \rightarrow S_{opt}) \otimes (S_{opt} \rightarrow \cdots \rightarrow a_n)$.

Fig. 2. SimPlanner algorithm

Problem Graph (PG). The first step of the algorithm is to build the PG, a graph inspired in a Graphplan-like expansion [2]. The PG may partially or totally encode the planning problem. The PG is a relaxed graph (delete effects are not considered) which alternates literal levels containing literal nodes and action levels containing action nodes.

The first level in the PG is the literal-level L_0 and it is formed by all literals in the initial situation a_0'. The PG creation terminates when a literal level containing all of the literals from the goal situation is reached in the graph or when no new actions can be applied. This type of relaxed graph is commonly

used in many heuristic search planners [4][5] as it allows to easily extract an approximate plan.

Necessary states. Second step is to compute the necessary state to execute each action in Π. A necessary state for an action a_i is the set of literals required to execute a_i and all its sucessors ($\mathsf{Succ}(a_i)$). In order to compute the necessary states, literals are propagated from the goal a_n to the corresponding action by means of the recursive formula shown in the algorithm.

Set of possible reachable states. This set comprises the necessary states to execute a set of parallel actions $\{a_i\}$ and all their successors $\mathsf{Succ}\{a_i\}$. A state will be definitely reachable if its literals make up a feasible situation in the new problem.

In a totally sequential plan Π, the possible reachable states will coincide with the necessary states to execute each action in Π. However, when Π comprises parallel actions it is necessary to compute the combinations of parallel actions, c_i, such that each element in c_i belongs to a different path from the current state to the goal state. In other words, c_i is a set of actions that can all be executed at the same time step. Consequently, the necessary state to execute actions in c_i denotes a state possibly reachable from $a_0{}'$.

Optimal state. In order to select the optimal reachable state, we define a heuristic function $f(x) = g(x) + h(x)$ associated to the cost of a minimal plan from the current situation $a_0{}'$ to the goal state a_n over all paths that are constrained to go through node x. $g(x)$ is the cost of the plan from x to a_n and is calculated straightforward from the original plan. $h(x)$ is the estimated length of an approximate plan P' from $a_0{}'$ to x.

```
Algorithm Approximated plan (a0', x) → plan P'
```

```
1. Build a ficticious action an' with preconds and no effects associated
to state x
2. P' = a0' → an'
2. L = x − Add(a0').
3. while L ≠ 0
   3.1 select l ∈ L
   3.2 select best a for l
   3.3 insert a in P'
   3.4 update L
4. return P'
```

Fig. 3. Outline of the algorithm to build an approximate plan

Step 3.1. Firstly, we form a set $UP \subset L$ with the unsolved preconditions of the nearest action a to $a_0{}'$. If $|UP| > 1$ then literals which appear later in the PG are removed from UP. If again $|UP| > 1$ we count the number of ways for solving each $l \in UP$ ($\{a \in A_i : l \in L_j \wedge l \in \mathsf{Add}(a) \wedge i <= j\}$), and select the literal with the lowest number of actions.

Step 3.2. The best action for $l \in \mathsf{Pre}(a)$ will be the action a_j which minimizes the number of *flaws* (preconditions not yet solved or preconditions of other actions which are deleted by a_j). To compute the number of flaws we have to check all possible positions of a_j in the plan provided that $a_j < a$.

Step 3.3. The new action a_j is inserted in the position obtained in the previous step. This position may be sequential - between two actions- or parallel to one or more actions. In this latter case, it must be possible to execute all actions in parallel, i.e. none of the actions will require and Add effect of another action or delete any of its preconditions. When this is not possible, actions must be executed sequentially.

The length of the returned plan P' will be the value of the heuristic function $h(x)$. Some of the properties of the heuristic function are:

- if x is an inconsistent or non-reachable state (all literals in x cannot be true at the same time), $h(x)$ returns ∞,
- if x is reachable from $a_0{}'$ so will be the states following x. This helpful information reduces vastly the cost of computing $h(x)$,
- although $h(x)$ is a non-admissible heuristic, it returns the optimal state for most of the test cases in empirical evaluations.

3.2 An Application Example

We will use an example of the *hanoi* domain to illustrate the behaviour of SimPlanner. The problem consists of four disks (huge **H**, big **B**, medium **M** and small **S**) and four pegs **P1**, **P2**, **P3** and **P4**. Initially, the four disks are on **P1** (Figure 4). The goal to be reached is shown in the final state of Figure 4. The unexpected situation occurs after executing the first action in the plan move S M P2 (move disk S from disk M to P2). At this time, a new smaller disk (tiny **T**) appears on **P4**. Figure 5 shows plan Π which is no longer executable because **P4** is not empty any more.

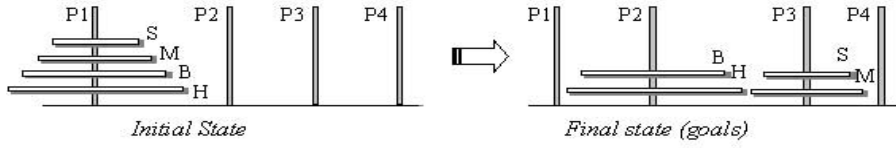

Fig. 4. Initial and final situation in the problem

Literals, denoted by numbers, are represented in Table 1. In Figure 5, literals above each node stand for the action preconditions and literals below each node represent the add and delete effects of the action.

The necessary states for each action in Π are shown in table 2. These are calculated by applying the formula in step 2 of SimPlanner algorithm. Then we compute the set of possible reachable states $RS = \{S(a_1), S(a_2, a_3), S(a_4), S(a_5), S(a_n)\}$.

Table 1. Literals in the *Hanoi* problem

3	on S P2	5	clear P3	6	on H P1	8	on B H	10	on M B	12	clear M
13	on T P4	15	clear S	16	clear T	19	clear P2	20	on M P3	21	clear B
23	clear P4	24	on S M	30	clear H	35	on B P4	39	clear P1	40	on H P2

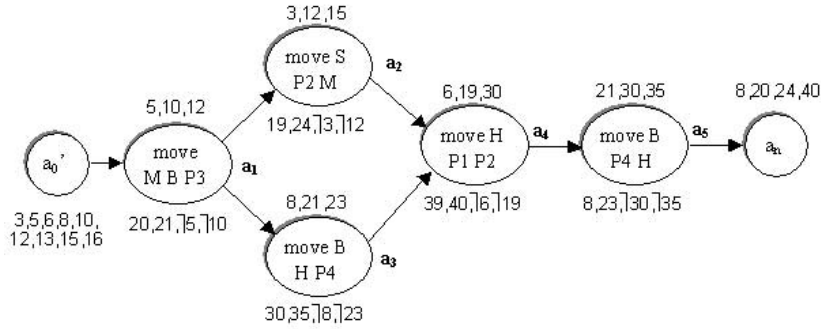

Fig. 5. Remaining plan to be executed (Π)

Notice $S(a_2)$ and $S(a_3)$ form a single state because both actions can be executed at the same time step. The result of $f(x)$ for each possible reachable state is shown in table 3. $h(RS1)$ returns ∞ because it is impossible to satisfy literals 12, 15, 5 and 23 due to the extra disk **T**. The same applies to $RS2$. However, $RS3$ is a reachable state because disk **T** can be located on top of disk **S** and clear S is not a condition required in $RS3$. The selected optimal state is $S(a_4)$ as this is the state that minimizes the value of $f(x)$. In case of equal values for $f(x)$ we will select that state with minimum value of $h(x)$.

Table 2. Necessary states for the hanoi example

Action	Description	Necessary state
a_n	final state	8, 20, 24, 40
a_5	move B P4 H	20, 21, 24, 30, 35, 40
a_4	move H P1 P2	6, 19, 20, 21, 24, 30, 35
a_3	move B H P4	6, 8, 20, 21, 23
a_2	move S P2 M	3, 6, 12, 15, 20, 21
a_1	move M B P3	3, 5, 6, 8, 10, 12, 15, 23

Finally, a plan from $a_0{}'$ to a_4 is built (t0: *current state*, t1: move M B P3, t2: move S P2 M, t3: move T P4 S, t4: move B H P4) and concatenated with the rest of the old plan from action a_4 (t5: move H P1 P2, t6: move B P4 H). For

Table 3. Reachability table for the hanoi problem

x	Pos. reachable state	$h(x)$	$g(x)$	$f(x)$
RS1	$S(a1)$	∞	5	∞
RS2	$S(a2) \cup S(a3)$	∞	3	∞
RS3	$S(a4)$	4	2	6
RS4	$S(a5)$	5	1	6
RS5	$S(a_n)$	6	0	6

our purposes, we have used the planner 4SP [6] as SimPlanner uses most of the data structures which are managed by this planner. The plan obtained when computing $h(x)$ is used as a lower bound in 4SP. For most of the test cases, the plan returned by 4SP was the same as the obtained in the computation of $h(x)$. In the *hanoi* problem, $h(x)$ returns the optimal plan in four execution steps.

4 Experimental Results

SimPlanner has been tested on several domains with different types of input information about external changes. The tested domains are *hanoi*, *monkey*, *blocksworld*, *logistics* and *mobile robots navigation*.

Figure 6 shows the comparative times for several problems between generating a complete plan from scratch or repairing only the affected parts of the plan (replanning process). In all except one of the test cases, the obtained plan was the optimal one. Temporal cost for replanning includes the cost of computing the reachable state plus time for generating the plan. As we can see in figure 6, replanning runtimes are much better when input changes are not very significant. In problems P5 and P7 respectively, the new current state forces to create a complete plan, so the cost of replanning is slightly higher.

We have also made the same tests with planner STAN2000 [7][1]. The time difference between planning and replanning is about the same proportion as the ones shown in Figure 6.

5 Conclusions

SimPlanner is a planning and execution system which allows the user to monitor the execution of a plan, interrupt the monitoring to input new information and repair the plan under execution according to unexpected event. SimPlanner performs an execution monitoring rather than simply testing the next action to execute. In this way, SimPlanner anticipates forthcoming situations and adjusts the plan in accordance.

The key point in SimPlanner is the replanning module. SimPlanner uses a graph-based planning approach supported by heuristic search techniques to efficiently replan in a dynamic world.

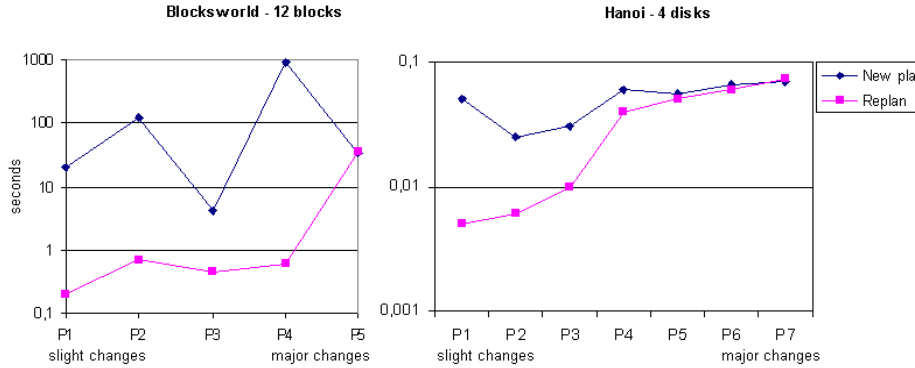

Fig. 6. Comparative results: generating a complete plan versus replanning

Currently, we are working on the integration of the planning algorithm and SimPlanner. The main objective is to be able to guarantee the optimality of the heuristic evaluation and improve the efficiency of the overall process. Additionally, we intend to extend SimPlanner to deal with time and consumable resources (as the battery in robot mobile environments).

References

1. Bacchus F.: AIPS 2000 competition results. Technical Report, University of Toronto, 2000. http://www.cs.toronto.edu/aips2000/.
2. Blum A. L., Furst M.L.: Fast Planning Through Planning Graph Analysis. Artificial Intelligence 90:281–300 (1997).
3. Despouys O., Ingrand F.F.: Propice-Plan: Toward a Unified Framework for Planning and Execution. European Conference on Planning 99, 280–292, 1999.
4. Haslum P., Geffner H.: Admissible heuristics for optimal planning. International Conference on AI Planning and Scheduling, 2000.
5. Hoffmann J., Nebel B.: The FF planning system: fast plan generation through heuristic search. Journal of Artificial Intelligence Research, 14:253–302, 2001.
6. Onaindia E., Sebastia L., Marzal E.: Incremental local search for planning problems. ECAI-2000 workshop on local search for planning & scheduling. Lecture Notes in AI, Springer-Verlag, 2001.
7. Fox M., Long D.: STAN public source code. `http://www.dur.ac.uk/CompSci/research/stanstuff/` (1999)
8. Stone P., Veloso M.: User-guided interleaving of Planning and Execution. Frontiers in Artificial Intelligence and Applications, 103-112, IOS Press, 1996.
9. Wilkins D.E.: Practical Planning: Extending the Classical AI Planning Paradigm. Morgan Kaufmann Publishers, San Mateo, CA, 1988.
10. Wilkins D.E., Myers K.L.: Asynchronous Dynamic Replanning in a Multiagent Planning Architecture. Advanced Planning Technology, 267–274, 1996.