RFF: A Robust, FF-Based MDP Planning Algorithm for Generating Policies with Low Probability of Failure

Florent Teichteil-Königsbuch and **Guillaume Infantes**
ONERA-DCSD
2 Avenue Édouard-Belin – BP 74025
31055 Toulouse Cedex 4
France

Ugur Kuter
University of Maryland
Department of Computer Science
College Park, MD 20742
USA

Introduction

Over the years, researchers have developed many efficient techniques, such as the planners FF (Hoffmann and Nebel 2001), LPG (Gerevini, Saetti, and Serina 2003), SATPLAN (Kautz, Selman, and Hoffmann 2006), SGPLAN (Hsu et al. 2006), and others, for planning in classical (i.e., deterministic) domains. Some of these planning techniques have been adapted for planning under uncertainty and provide some impressive performance results. For example, the FF-REPLAN (Yoon, Fern, and Givan 2007) is a reactive online planning algorithm that has been demonstrated to be very effective for many MDP planning problems. As another example, planning-graph techniques inspired by FF has also been generalized to planning under nondeterminism and partial observability (Hoffmann and Brafman 2005; Bryce, Kambhampati, and Smith 2006). Finally, the conformant-planning approach of (Palacios and Geffner 2006; 2007) describes how to translate any conformant planning problem into a classical problem for a classical planner. Their approach generates a single translation, and can only find conformant solutions.

In this paper, we describe our work that investigates a somewhat middle-ground between the previous approaches described above. In particular, we present an MDP planning algorithm, called RFF, for generating offline robust policies in probabilistic domains. Like both of the above approaches, RFF first creates a relaxation of an MDP planning problem by translating it into a deterministic planning problem in which each action corresponds to an effect of a probabilistic action in the original MDP and for every such effect in the original MDP there is a deterministic action, and there are no probabilities, costs, and rewards. In this relaxed planning problem, RFF computes a policy by generating successive execution paths leading to the goal from the initial states by using FF. The policy returned by RFF has a low probability of failing. In our approach, we interpret this not as the probability of reaching a goal but as the probability of causing any replanning during execution.

In this work, we use a Monte-Carlo simulation in order to compute the probability that a partial policy would fail during execution (i.e., the probability that the execution of the policy will end in a state for which the policy does not specify any action). Based on this probability, RFF decides whether to expand the policy further, or return it as a solution for the input MDP.

Similar to FF-REPLAN, which was the winner of the International Probabilistic Planning Competition in 2004, RFF is a replanning algorithm for MDPs. FF-REPLAN is an on-line probabilistic planner which computes a sequence of actions from a given state. This sequence of actions is a *plan*, i.e. a list of $(state, action)$ pairs which indicates the action to perform in each successive state in order to reach the goal. Since actions' effects are probabilistic, the sequence of actions is not guaranteed to succeed: each time the plan's execution leads to a state which is not in the plan, FF-REPLAN has to recompute a new plan from scratch in the current state.

Unlike FF-REPLAN, however, RFF generates offline policies that have some guarantees for not failing; if a failure still occurs during execution, then RFF updates its previous policy to circumvent the failure. It is similar to CONTINGENT-FF mentioned above in its way to use FF to generate offline policies using relaxations of the planning problems.

Another difference between FF-REPLAN and RFF is that the latter one does not compute the same plan over and over again, if an undesirable action outcome occurs in which a plan was already computed in the past (in the current execution or in previous executions).

Like FF-REPLAN, RFF is not guaranteed to generate optimal solutions to MDP planning problems and does not currently handle the dead-end states. In our experiments, RFF generated optimal solutions in some cases and near-optimal solutions in most. In our discussions below, we also describe some exploratory ideas on dealing with these issues.

The RFF Algorithm

The main idea behind RFF is to generate an initial path from the initial state to some goal state by using the classical planner FF (Hoffmann and Nebel 2001) on a relaxation of the MDP. Since this path is deterministic, its execution can fail with a high probability in the sense that the execution of this path can lead to a state for which the path does not specify any actions. We call such states *failure states* because they represent states where the deterministic plan failed. In each failure state, we compute a new path to the some goal state by using again FF from this failure state. Then, we look for new failure states from which we compute new deterministic paths, and so on. We stop as soon as the probability to reach some failure state from the initial state is less than a given threshold. As a result, RFF produces a robust policy to goal states with a controlled failure probability.

The RFF planner used in the competition is based on a graph implementation defined as:

- a graph is a mapping from PPDDL states to nodes ;
- a PPDDL state is a set of true predicates ;
- a node is a tuple containing the current best action and best value in this state, as well as a mapping from applicable actions in this state to a list of all stochastic outcomes (edges) for each action ;
- an edge (outcome) is a tuple containing the probability, the reward, and the next state of the outcome.

A node in G is a *terminal node* if it does not have any outgoing edges.

RFF's pseudo-code is shown in Algorithm 1. In lines 1 to 2, RFF generates a deterministic relaxation of the input MDP and the initial policy is the empty policy. This means that initially, the graph G contains a single state representing the initial state I. There are several known strategies for generating a deterministic relaxation of an MDP: see (Yoon, Fern, and Givan 2007) for a review of some of them. In this work, we generate a deterministic relaxation by substituting all stochastic outcomes of an action by the most probable outcome. This strategy is appropriate to our approach because we seek to minimize path execution failure during execution due to least probable outcomes.

In lines 3 to 26, RFF computes successive paths from each failure state, until the probability to reach some failure state is less than a given threshold ρ (line 26). In lines 5 and 6, the planner first generates a deterministic problem whose initial state is the current terminal state s, and then it invokes FF.

Next RFF generates the state trajectory by successively applying the original probabilistic versions of the actions generated by FF in the current terminal state. This process also computes new terminal states and RFF keeps them in the set $\mathcal{T}$ in Algorithm 1. In lines 9 to 12, RFF adds the current path generated by FF into the partial policy π. Then, RFF computes the set of terminal states of the graph G (lines 13 – 21). Before going into the next iteration, RFF also computes the probability to reach a failure state from the initial state by following the current policy (line 25). As this computation can be costly, we present in the next section a sequential Monte-Carlo method we used to statistically assess this probability.

Sequential Monte-Carlo Simulation for Computing Policy Failure Probabilities

In line 25 of Algorithm 1, we need to compute the probability to reach some failure states from the initial state by applying the current policy in the graph. Theoretically, this probability can be iteratively assessed by computing the successive probabilities $P_t(\mathcal{T} \mid s)$ of reaching some terminal states in $\mathcal{T}$ from a state s in t steps:

- $P_0(\mathcal{T} \mid s) = \delta_{\mathcal{T}}(s)$
- $P_t(\mathcal{T} \mid s) = \sum_{s' \in G.edges(s)} P(s' \mid s, \pi(s)) P_{t-1}(\mathcal{T} \mid s')$

The probability to reach some terminal states from the initial state I is then: $\lim_{t \to +\infty} P_t(\mathcal{T} \mid I)$.

Nevertheless, this computation may be quite costly, so that we assess this probability by means of sequential Monte-Carlo simulation instead. The principle is very simple and quite efficient: it consists in simulating N trajectories which all start from the initial state, and which end each as soon as a terminal state is reached or after a given depth. The probability to reach some terminal state is approximately the ratio between the number of trajectories which reach a terminal state and N. The sequential Monte-Carlo simulation, performed by the $compute\_failure$ subroutine in Algorithm 2, computes the probability to reach terminal states. In line 7, a sampled outcome of a state s by

Algorithm 1: RFF

```
   // $M$: the input MDP
   // $I$: initial state
   // $G = (nodes\_map)$: explored states
   // $nodes\_map = state \rightsquigarrow (action, value, edges\_map)$
   // $edges\_map = \mathcal{A}_T(state) \rightsquigarrow edges\_list$
   // $edge = (probability, node)$
   // $\mathcal{T}$: set of tip nodes
   // $failure\_probability$: probability to reach
      a terminal state
   // $0 \leqslant \rho < 1$: failure probability
      threshold
1  $\mathcal{D} \leftarrow generate\_deterministic\_domain(M)$;
2  $\mathcal{T} \leftarrow \{I\}$;
3  repeat
4      for $s \in \mathcal{T}$ do
5          $\mathcal{P}_s \leftarrow generate\_deterministic\_problem(s)$;
6          $\{(s_1, a_1), \cdots, (s_p, a_p)\} \leftarrow$ FF$(\mathcal{D}, \mathcal{P}_s)$;
7          for $1 \leqslant i \leqslant p$ do
8              if $s_i \notin G.nodes$ or $s_i \in \mathcal{T}$ then
9                  $G.nodes.add(s_i)$;
10                 $em \leftarrow outcomes(s_i)$;
11                 $G.nodes(s_i).insert(em)$;
12                 $\mathcal{T}.remove(s_i)$;
13                 for $a \in applicable(s)$ do
14                     $em \leftarrow G.nodes(s_i).edges\_map(a)$;
15                     for $e \in em$ do
16                         if $e.state \notin G.nodes$ then
17                             $G.nodes.add(e.state, \texttt{nil})$;
18                             $\mathcal{T}.add(e.state)$;
19                         end
20                     end
21                 end
22             end
23         end
24     end
25     $failure\_probability \leftarrow compute\_failure(I, G)$;
26 until $failure\_probability \leqslant \rho$ ;
```

applying the current policy in s, is obtained by sampling the outcomes of s in the graph G.

Algorithm 2: $compute\_failure$ function

```
Input: $I$ // initial state
    $\mathcal{G}$ // set of goal states
    $\mathcal{T}$ // set of terminal states
    $G$ // explored states graph
    $N$ // number of simulations
    $depth$ // maximum depth of simulated
    trajectories
Output: $\varphi$ // probability to reach a
    terminal state (failure probability)
1  $\varphi \leftarrow 0$;
2  for $1 \leqslant i \leqslant N$ do
3      $s \leftarrow I$;
4      $cnt \leftarrow 0$;
5      repeat
6          $cnt \leftarrow cnt + 1$;
7          $s \leftarrow sample\_outcome(s, G)$;
8          if $s \in \mathcal{T}$ then
9              $\varphi \leftarrow \varphi + \frac{1}{N}$;
10         end
11     until $cnt \leqslant depth$ and $s \notin \mathcal{G}$ and $s \notin \mathcal{T}$ ;
12 end
```

Choosing the Goal States for FF

Different strategies can be applied to compute paths from failure states depending on the deterministic problem given to FF. In the previous subsection, we implicitly presented the strategy which consists in computing a path from a given failure state to the goal states.

Although this strategy's implementation is very simple, it often results in computing several times the same subpaths. Let us assume FF computes a path $\{(s_1, a_1), \cdots, (s_p, a_p)\}$ after having already computed a lot of paths. It is quite probable there exists $1 \leqslant k \leqslant p$ where the subpath $\{(s_k, a_k), \cdots, (s_p, a_p)\}$ was already computed in previous paths' computations.

To avoid this drawback, we also developed two other strategies. In both of them, when RFF generates a deterministic problem for FF, the goals of that problem is a subset of states for which the policy is already computed. In this case, one expects FF to reach neighbor states for which the policy is available, instead of trying to reach perhaps far goal states. The two strategies differ in how the set of goal states for FF is generated, as described below.

In the first strategy for choosing goals for FF, we randomly choose k many states from the policy graph and give them as goals to FF. The second strategy is a Monte-Carlo simulation that starts the terminal state s in which we invoke FF and ends in the states that are already in the policy. We perform k simulation trials to generate k most likely states that are reachable from s. We refer to the first strategy as RFF-RG (RFF-RANDOMGOAL for "reach randomly-selected goals"), and to the second one as RFF-MC (RFF-MONTE CARLO for "reach goals selected by the Monte-Carlo simulation').

Discussion and Ongoing Work

RFF is not optimal because it does not optimize the accumulated rewards, known as the *value function* in Markov Decision Processes (Puterman 1994). The value function in the initial state is the average value of accumulated rewards along all trajectories which start from the initial state.

As a result, we can sub-optimize the value function by computing deterministic paths along which the reward fluent is optimized. The average accumulated rewards won't most probably be optimal, but its value may be greater than the value of paths generated with plain FF. Optimized deterministic paths can be computed with METRIC-FF (Hoffmann 2003).

Acknowledgments. This research was supported in part by the French *Délégation générale pour l'armement* grant 07.60.031.00.470.75.01., DARPA's Transfer Learning and Integrated Learning programs, and NSF grant IIS0412812. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

Bryce, D.; Kambhampati, S.; and Smith, D. E. 2006. Planning Graph Heuristics for Belief Space Search. *Journal of Artificial Intelligence Research* 26:35–99.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through Stochastic Local Search and Temporal Action Graphs. *JAIR* 20:239–290.

Hoffmann, J., and Brafman, R. 2005. Contingent Planning via Heuristic Forward Search with Implicit Belief States. In *ICAPS-05*.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2003. The metric-FF planning system: Translating ignoring delete lists to numerical state variables. *Journal of Artificial Intelligence Research* 20.

Hsu, C. W.; Wah, B. W.; Huang, R.; and Chen, Y. X. 2006. New Features in SGPlan for Handling Soft Constraints and Goal Preferences in PDDL3.0.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SatPlan: Planning as Satisfiability.

Palacios, H., and Geffner, H. 2006. Compiling Uncertainty Away: Solving Conformant Planning Problems Using a Classical Planner (Sometimes). In *AAAI*.

Palacios, H., and Geffner, H. 2007. From Conformant into Classical Planning: Efficient Translations that may be Complete Too. In *ICAPS-07*.

Puterman, M. L. 1994. *Markov Decision Processes*. John Wiley & Sons, INC.

Yoon, S.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *In proceedings of the 17th International Conference on Automated Planning and Scheduling*.