Monitoring Policy Execution

Christian Fritz and **Sheila A. McIlraith**
Department of Computer Science, University of Toronto, Toronto, ON M5S 3G4, CANADA
{fritz,sheila}@cs.toronto.edu

Abstract

In this paper we explore the paradigm of planning and execution in stochastic domains with (a) exogenous events, (b) an incorrect model, and/or (c) incomplete forward search (i.e. discarding low probability outcomes). We consider forward search-based planning algorithms for Markov Decision Processes (MDPs), exploring the reachable state space from a given initial state. Under such circumstances, an executing policy often finds itself in an unexpected state, bringing into question the continued optimality of the policy being executed (or near-optimality in the case where the optimal policy was approximated). Replanning in this unexpected state is a naive and costly solution that is often, we argue, unnecessary. In this paper we exploit regression to identify the subset of each expected state that is critical to the optimality of a policy. With this information in hand, we can often avoid replanning when faced with an unexpected state. Our analysis offers theoretic bounds on optimality in certain cases as well as empirical results demonstrating significant computational savings compared to replanning from scratch.

1 Introduction

The de-facto standard for modeling decision making in stochastic domains is Markov Decision Processes (MDPs). In most cases, MDPs are solved off-line by creating a policy that maps each state to an action that maximizes the expected accumulated reward over time. With this policy in hand, an agent knows how to act optimally in any state of the world. Unfortunately, solving an MDP in such a way is computation-intensive and impossible for large or infinite state spaces, or when time is limited. As such, a reasonable alternative is to compute an approximately optimal policy via state-based search from a known initial state. This generates a policy for (a subset of) the state space that is accessible from the current state within a bounded number of actions. Two questions arise from such an approach: how close to optimal is the resulting policy, and how robust is it.

In this paper we address the issue of policy robustness. Since the MDP is only solved for a subset of the state space an agent can find itself outside this subset during execution of the policy. This can happen for at least two reasons: i) the initial search ignored less likely outcomes of actions or ignored the possible occurrence of some unlikely exogenous events, or ii) the transition function erroneously neglected the possibility of certain outcomes or events. This is particularly true of sampling-based approaches, sometimes used to find an approximately optimal policy for a problem defined over a continuous state space (e.g., [4]). Regardless of the cause, a discrepancy between actual and anticipated outcomes requires the agent to decide how to proceed. A common response is to replan starting from the actual current state. We argue that in many cases of discrepancy detection it is not necessary to replan because the aspect of the state that is deviant has no bearing on the optimality of the remaining policy execution.

To address this problem, we develop an approach to monitoring policy execution that is inspired by a technique commonly used to monitor the execution of deterministic plans, dating as far back as PLANEX ([6]), Shakey's execution strategy. In PLANEX, each step of a (straight-line) plan was annotated with the regression [7] of the goal together with the remaining preconditions for the remainder of the plan. The regressed formulae captured those conditions that had to be true at each step of the plan in order to ensure that remaining plan was still executable and still led to the goal. If discrepancies arose during execution, these annotations could be verified to determine whether the plan was still executable and valid. We propose to proceed similarly for decision-tree search, replacing the goal with a heuristic approximation of the real value function, and, since optimality is relative rather than absolute, also considering alternatives. Experiments with an implementation of our approach illustrate its potential to drastically speed-up the decision to continue execution of a policy or to abort and replan. While our approach is described in the situation calculus, it is amenable to use with any action description language for which regression can be defined (e.g., STRIPS and ADL).

In Section 2 we review the situation calculus representation of relational MDPs, and their solution using decision-tree search. In Section 3 we show how to annotate a policy and present an algorithm that exploits this annotation to determine continued optimality during policy execution. Next we analyze properties of our approach including the space complexity of our annotation. We also consider two particular classes of MDPs: finite horizon MDPs, and large MDPs that are solved approximately via sampling. In both cases we establish the continued optimality of a policy following execution of our monitoring algorithm. We conclude with a discussion of related and future work.

In [3] we considered the case of monitoring optimality for deterministic plans. This work extends that approach to decision-theoretic planning.

2 Background

2.1 Situation Calculus

The situation calculus is a logical language for specifying and reasoning about dynamical systems [7]. In the situation calculus, the *state* of the world is expressed in terms of fluents, $\mathcal{F}$, predicates relativized to a particular *situation* s, e.g., *hasCoffee*$(\vec{x}, s)$. A situation s is a *history* of the primitive actions performed from a distinguished initial situation S_0. The function $do(a, s)$ maps an action and a situation into a new situation thus inducing a tree of situations rooted in S_0, $s \sqsubset s'$ states that s precedes s', i.e. $s' = do(\vec{a}, s)$ for some sequence of actions $\vec{a}$.[1]

A basic action theory in the situation calculus, $\mathcal{D}$, comprises four *domain-independent foundational axioms*, and a set of *domain-dependent axioms*. Details of the form of these axioms can be found in [7]. Included in the domain-dependent axioms are axioms specifying what is true in the initial state S_0, action precondition axioms *Poss*$(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s)$ one for each action a, where $\Pi_a(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$, and successor state axioms (SSAs). The latter are axioms of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, one for each fluent F. $\Phi_F(\vec{x}, a, s)$ characterizes the truth value of the fluent F in the situation $do(a, s)$ in terms of what is true in the current situation s.

Regression

The regression of a formula ψ through an action a is a formula ψ' that holds prior to a being performed if and only if ψ holds after a is performed. In the situation calculus, regression is defined inductively using the successor state axiom for F as above:

$$\mathcal{R}[F(\vec{x}, do(a, s))] = \Phi_F(\vec{x}, a, s)$$
$$\mathcal{R}[\neg\psi] = \neg\mathcal{R}[\psi]$$
$$\mathcal{R}[\psi_1 \wedge \psi_2] = \mathcal{R}[\psi_1] \wedge \mathcal{R}[\psi_2]$$
$$\mathcal{R}[(\exists x)\psi] = (\exists x)\mathcal{R}[\psi]$$

We denote the repeated regression of a formula $\psi(do(\vec{a}, s))$ back to a particular situation s by $\mathcal{R}_s$, e.g. $\mathcal{R}_s[\psi(do([a_1, a_2], s))] = \mathcal{R}[\mathcal{R}[\psi(do([a_1, a_2], s))]]$. Intuitively, the regression of a formula ψ over an action sequence $\vec{a}$ is the condition that has to hold now for ψ to hold after executing $\vec{a}$. It is predominantly comprised of the fluents that play a role in the conditional effects of the actions in the sequence. Regression is a purely syntactic operation. Nevertheless, it is often beneficial to simplify the resulting formula for later evaluation. Regression can be defined in many action specification languages (e.g. STRIPS, ADL).

Notation: Lower case letters denote variables in the theory of the situation calculus, upper case letters denote constants. However, we use capital S to denote arbitrary but explicit situation terms, that is $S = do(\vec{a}, S_0)$ for some explicit action sequence $\vec{a}$. For instance we will use, S_k for $k > 0$ to denote a situation expected during planning, and S^* to denote the actual situation that arises during execution. Variables that appear free are implicitly universally quantified unless stated otherwise.

[1] For readability action and fluent arguments are generally suppressed. Also, $do(a_n, do(a_{n-1}, \ldots do(a_1, s)))$ is abbreviated to $do([a_1, \ldots, a_n], s)$ or $do(\vec{a}, s)$.

2.2 Representing and Solving MDPs

An MDP is described through a state space $\mathcal{S}$, a set of actions $\mathcal{A}$, a transition function $\mathcal{T}$, with $\mathcal{T}(s, a, \cdot)$ denoting a distribution over $\mathcal{S}$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, a reward function $\mathcal{R} : \mathcal{S} \to I\!R$, and a cost function $\mathcal{C} : \mathcal{A} \times \mathcal{S} \to I\!R$. The state is assumed to be fully-observable by an agent acting in such an environment and the agent's goal is to behave according to a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the *value function* defined using the infinite horizon, discounted reward criterion, defined in terms of the expectation $\mathcal{E}$:

$$V^\pi(s) = \mathcal{E}\left[\sum_{i=0}^{\infty} \gamma^i (r_i - c_i) \,\middle|\, s, \pi\right]$$

where r_i is the reward obtained after performing policy π for i steps starting in s, c_i is the cost incurred by the action performed at that stage, and $0 \leq \gamma < 1$ is the discount factor. Similarly, the *Q-function* is defined as

$$Q^\pi(a, s) = R(s) - C(a, s) + \gamma \mathcal{E}_{s' \sim T(s,a,\cdot)}\left[V^\pi(s')\right]$$

where the expectation $\mathcal{E}$ is over the transition probabilities $T(s, a, \cdot)$. Finally, the optimal value function and optimal Q-function are defined as $V^*(s) = \sup_\pi V^\pi(s)$ and $Q^*(a, s) = \sup_\pi Q^\pi(a, s)$ respectively. The optimal policy behaves greedily w.r.t. to Q^*, i.e. $\pi^*(s) = \text{argmax}_a\, Q^*(a, s)$.

An MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{C} \rangle$ can be represented through a basic action theory $\mathcal{D}$ in the situation calculus as follows (cf. [7, 2]): Fluents describe the set of states relationally, thus $\mathcal{S}$ is the set of all, possibly infinite, combinations of fluent values. Further the user specifies for each (stochastic) action $a \in \mathcal{A}$ a predicate *Choice*(a, a'_i) describing a collection of primitive actions a'_i that form the unique outcomes of executing a. Preconditions are defined for a, but successor state axioms are defined in terms of the primitive actions describing the outcomes. Using predicate *Prob*(a'_i, a, p, s), the user specifies the probability p for outcome a'_i when a is performed in situation s. *Prob* and *Choice* describe the transition probabilities $\mathcal{T}$. The specification is completed by two more predicates *Reward*(r, s) describing the reward r obtained in situation s, and *Cost*(a, c, s) describing the cost c of primitive action a in s. The later deviates slightly from the definitions used in [7, 2] as it defines costs for the outcomes of stochastic actions rather than the stochastic actions themselves. This is slightly more expressive and contains the alternative as a special case.

The relational representation is more efficient than simple state enumeration but more importantly it also allows us to regress preconditions, rewards, costs, and probabilities over actions. This is the key requirement for our algorithm.

Solving MDPs through Search

We assume we are given a decision-tree search planner, as described e.g. in [5], which operates as follows. Starting with a search tree containing only one node labeled with situation S_0, describing the current state of the world, the planner works by repeatedly expanding nodes in the tree and adding their successors (Figure 2 shows an example tree). A node labeled with situation s, denoted $N[s]$, has successors $N[a_i, s]$, labeled with actions $a_i \in \mathcal{A}$. If a_i is possible in s, i.e. $\mathcal{D} \models$ *Poss*(a_i, s), then $N[a_i, s]$ has successors labeled with the possible successor situations

$do(a'_{i1}, s), \ldots, do(a'_{im}, s)$ where the a'_{ij} are the outcomes (nature's choices) of action a_i as defined by *Choice*(a_i, a'_{ij}). Situation labeled nodes $N[s]$ have an associated reward r as defined through *Reward*(r, s), edges $E[a', s]$ denoting action outcomes a' have associated costs c and probabilities p, defined resp. by *Cost*(a', c, s) and *Prob*(a', a, p, s). We make no assumptions regarding the expansion strategy of the planner, that is, how to choose the next node to expand, nor about its cutoff criterion, used to determine when to stop expanding. But we do assume the existence of a predicate $\bar{V}(v, s)$ that provides a heuristic estimate v of the value of the optimal value function (V^*) in situation s.[2] This heuristic is used to estimate the value in all leaf nodes.

Given a tree spanned this way, we can obtain a better estimate of the real value function for situations in the tree, by backing-up the values from the leaves to the root using the standard update rules:

$$Q(a, s) = r_s + \sum_{a'} p_{sa'} \left(\gamma V(do(a', s)) - c_{sa'}\right)$$

$$V(s) = \max_{a \in \mathcal{A}(s)} (Q(a, s))$$

with $r_s, p_{sa'}, c_{sa'}$ s.t. $\mathcal{D} \models$ *Reward*$(r_s, s) \wedge$ *Prob*$(a', a, p_{sa'}, s) \wedge$ *Cost*$(a', c_{sa'}, s)$, and initialization in the leaves $V(s) = v$, s.t. $\mathcal{D} \models \bar{V}(v, s)$. This function converges to the real value function as the search horizon increases, that is, the farther the search is performed, the closer the approximation will be. The best action to take in the initial situation S_0 according to this function is the greedy action $a^* = argmax_{a \in \mathcal{A}}(Q(a, S_0))$, and similarly for all subsequent actions. This produces a conditional plan representing the best (partial) policy according to this value function approximation, starting in S_0, of the form

$$\begin{aligned}\pi(S_0) = a;\ &\textbf{if } \varphi(a'_1) \textbf{ then } \pi(do(a'_1, S_0))\\ &\textbf{elseif } \varphi(a'_2) \textbf{ then } \pi(do(a'_2, S_0))\\ &\ldots\\ &\textbf{elseif } \varphi(a'_m) \textbf{ then } \pi(do(a'_m, S_0)) \textbf{ fi}\end{aligned}$$

where the $\{\pi(do(a'_i, S_0))\}_{1 \le i \le m}$ denote the sub-policies for the considered outcomes of action a, and $\varphi(a'_i)$ denotes a formula that holds iff the outcome a'_i of a has happened. Following the assumption of full-observability in the MDPs we consider, these formulae can always be evaluated. But what if after performing action a in S_0 we do not end in either of these situations $do(a'_i, S_0)$ but in some unexpected situation S^*, or after planning we observe that some exogenous event has altered the world and we are no longer in S_0 but in S^*? It seems one would need to perform time consuming replanning starting from S^* in these cases. We argue that this can often be avoided, namely when the discrepancy between an expected state and the actual state is irrelevant to the remaining policy. To distinguish between relevant and irrelevant discrepancies, we propose, roughly, to regress the value function and other relevant information over the policy to derive a condition for its optimality (or near-optimality depending on the initial policy's degree of optimality) in terms of the current situation.

We assume that the planner not only returns the policy, but also the search tree itself. This serves our approach and also

[2] cf. e.g. [5] for notes on how to obtain such a heuristic

$(\pi(S_0), T(S_0)) \leftarrow$ *plan*(S_0)
$\hat{\pi}(S_0) \leftarrow$ *annotate*$(\pi(S_0), T(S_0))$
loop
  obtain S^*
  if $S^* \in T(S_0)$ ***then*** $\{\ \hat{\pi} \leftarrow \hat{\pi}(S^*)\ \}$
  else $\{$ *choose* $S' \in T(S_0)$*;* $\hat{\pi} \leftarrow$ *patch*$(\hat{\pi}(S'), S^*)\ \}$
  extend$(\hat{\pi})$
  execute-next-action$(\hat{\pi})$

Figure 1: general flow of control

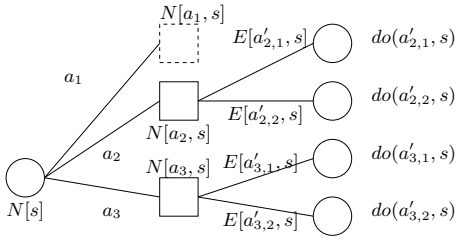


Figure 2: A sample search tree. Circles denote states, boxes denote nature's choices.

enables the planner to further improve the remaining policy during execution.

3 Monitoring Policy Execution

Given a planner that provides a (near-)optimal policy according to the given heuristic function as described above starting from situation S, we assume, roughly, the general control flow of Figure 1, presented in pseudo-code, where $T(S)$ is the (sub-)search-tree from situation S. That is, first the planner generates a (near-) optimal policy and also returns the search tree. This policy is annotated by associating with each step an annotated copy of the corresponding sub-search tree (Section 3.1). Then the policy is executed (cf. loop). During execution the policy may be further improved by extending the current sub-search tree. However, if the current actual situation S^* is unexpected and in particular not planned for, for reasons outlined above, we first need to patch the policy from some (expected) situation S' included in the original search tree to reflect this discrepancy (Section 3.2). We will later address the issue of how to choose S', but intuitively it should be the result of one of the considered outcomes or S itself. It turns out that patching a policy by patching its associated sub-search tree and possibly extracting a new greedy policy can generally be done much faster than replanning from the new situation when exploiting both, the annotations we propose, and knowledge about the discrepancy itself. This is described in Section 3.2.

3.1 Annotation

We annotate each sub-policy with a copy of the (sub-)search tree of the corresponding node of the overall search tree returned by the planner as follows. Recall that $N[S]$ denotes the (unique) node in the search tree labeled with situation S, $N[a, S]$ its successor nodes representing the execution of action a in S, and $N[do(a'_i, S)]$ the node representing its i^{th} outcome. $E[a'_i, S]$ denotes the edge between $N[a, S]$ and $N[do(a'_i, S)]$.

Definition 1 (Annotated Search Tree)**.** The *annotated search tree* $\hat{T}(S) = (\hat{\mathbf{N}}, \hat{\mathbf{E}})$ for $T(S) = (\mathbf{N}, \mathbf{E})$ is defined as follows for all considered descendants S' of S:

$$\hat{N}[S'] = \begin{cases} (\mathcal{R}_S[\bar{V}(v, S')], v') & \text{if leaf node} \\ (\mathcal{R}_S[\textit{Reward}(r, S')], r') & \text{otherwise} \end{cases}$$

with v' s.t. $\mathcal{D} \models \bar{V}(v', S')$, and r' s.t. $\mathcal{D} \models \textit{Reward}(r', S')$,

$$\hat{N}[a, S'] = (\mathcal{R}_S[\textit{Poss}(a, S')], x)$$

with x s.t. $\mathcal{D} \models (\textit{Poss}(a, S') \wedge x = 1) \vee (\neg\textit{Poss}(a, S') \wedge x = 0)$, and

$$\begin{aligned} \hat{E}[a', S'] \quad = \quad & \big((\mathcal{R}_S[\textit{Cost}(a', c, S')], c'), \\ & (\mathcal{R}_S[\textit{Prob}(a', a, p, S')], p')\big) \end{aligned}$$

with c' s.t. $\mathcal{D} \models \textit{Cost}(a', c', S')$, and p' s.t. $\mathcal{D} \models \textit{Prob}(a', a, p', S')$. Nodes have further an associated value:

$$\hat{N}[S'].v = \begin{cases} v' & \text{if leaf node} \\ r' + \max_a(\hat{N}[a, S'].v) & \text{otherwise} \end{cases}$$

$$\hat{N}[a, S'].v = \sum_{a'} \hat{E}[a', S'].p'(\gamma \cdot \hat{N}[do(a', S')].v - \hat{E}[a', S'].c')$$

where v', r' are as above and $\hat{E}[a', S'].p'$ and $\hat{E}[a', S'].c'$ denote the resp. values annotated in $\hat{E}[a', S']$.

That is, we annotate each situation-node of the tree with the regression of the heuristic estimate of the real value function if it is a leaf node, or the regression of the reward predicate otherwise. Action nodes are annotated with the regression of their preconditions, and edges with the regression of their respective cost- and probability-predicates. In all cases, the regression is performed back to the situation labeling the root of the tree, S. The regressed predicates are all the information needed to determine the optimality of the policy given the search horizon, solely based on what is true in the current situation. This allows for greatly improved reevaluation if on-line we find ourselves in the unexpected situation S^* rather than S: *Only those annotated conditions that mention fluents that are affected by the discrepancy between S^* and S need to be reevaluated.* Given that most discrepancies only affect a fraction of all fluents, this saves a lot of computation. After reevaluating the affected conditions, the values of Q-function and value function can be updated (back-up). This on-line behavior is subject of the next section.

Intuitively, the regression of a function over a sequence of actions $\vec{a}$ describes in terms of the current situation, the value the function will take after performing $\vec{a}$. As an example, consider the task of delivering a package to a location. Assume the heuristic function yields a value $v = 0$ when a truck containing the package is at the right location and $v = 1$ otherwise. Then, regressing this function through the action of driving the truck to the right location would yield a formula stating "$v = 0$ *(v will be 0) if the package is on the truck, and $v = 1$ otherwise*".

Note the correspondence between $\hat{N}[S'].v$ and the value function $V(S')$, and between $\hat{N}[a, S'].v$ and the Q-function $Q(a, S')$.

3.2 Execution Monitoring

Assume we have executed a (possible empty) prefix $a_1, a_2, \ldots, a_k$ of an earlier, in situation S, generated policy and the outcomes of these actions were $a'_{1i_1}, a'_{2i_2}, \ldots, a'_{ki_k}$ so that we expect to be in situation $S_k = do([a'_{1i_1}, a'_{2i_2}, \ldots, a'_{ki_k}], S)$, but in fact find ourselves in a different situation S^* that the current search tree and policy don't account for. The naive solution is to either fall back to greedy behavior according to the heuristic estimate of the value function or to replan from S^*. We here propose a better solution, "patching" the search tree to reflect all values with respect to S^* in place of S_k. We will do this as outlined earlier, by reevaluating all conditions that are affected by the discrepancy and backing-up the altered values to also update the Q-function and value function values.

Let $\Delta_F(S_k, S^*)$ be the set of fluents whose truth values differ between S_k and S^*, i.e. $\Delta_F(S_k, S^*) = \{F(\vec{X}) \mid F \in \mathcal{F}$ and $\mathcal{D} \models F(\vec{X}, S_k) \not\equiv F(\vec{X}, S^*)\}$, with $\mathcal{F}$ the set of fluents[3]. Only conditions mentioning any of these fluents need to be reevaluated, all others remain unaffected by the discrepancy. Let *fluents*(ϕ) denote the fluents occurring in formula ϕ.

```
redoSit( N̂[S'], T̂(S_k), S* )
  Φ ← successors(N̂[S'], T̂(S_k))
  if (Φ = ∅) then
    if (fluents(R_S[V̄(v, S')]) ∩ Δ_f = ∅) then N̂[S'].v ← N̂[S'].v'
    else N̂[S'].v ← v_new s.t. D ⊨ R_S[V̄(v_new, S')](S*)
  else
    maxq ← max_{N̂[a,S'] ∈ Φ}(redoAction(N̂[a, S'], T̂(S_k)))
    if (fluents(R_S[Reward(r, S')]) ∩ Δ_f = ∅) then r' ← N̂[S'].r'
    else r' ← r_new s.t. D ⊨ R_S[Reward(r_new, S')](S*)
    N̂[S'].v ← r' + maxq
  return N̂[S'].v

redoAction( N̂[a, S'], T̂(S_k), S* )
  Φ ← successors(N̂[a, S'], T̂(S_k))
  if (fluents(R_S[Poss(a, S')]) ∩ Δ_f = ∅) then x ← N̂[a, S'].x
  else x ← x_new s.t.
       D ⊨ (R_S[Poss(a, S')](S*) ∧ x_new = 1) ∨
            (¬R_S[Poss(a, S')](S*) ∧ x_new = 0)
  if (x = 0) then N̂[a, S'].v ← −∞
  else if(Φ = ∅) then N̂[a, S'].v = v_new s.t. D ⊨ R_S[V̄(v_new, S')](S*)
    else
      foreach (N̂[do(a', S')] ∈ Φ) do redoOutcome(Ê[a', S'])
      N̂[a, S'].v ← γ Σ_{a'} Ê[a', S'].p'(Ê[a', S'].c' + N̂[do(a', S')].v)
  return N̂[a, S'].v

redoOutcome( Ê[a', S'], T̂(S_k), S* )
  if (fluents(R_S[Cost(a', c, S')]) ∩ Δ_f = ∅) then c' ← Ê[a', S'].c'
  else c' ← c'_new s.t. D ⊨ R_S[Cost(a', c'_new, S')](S*)
  if (fluents(R_S[Prob(a', a, p, S')]) ∩ Δ_f = ∅) then p' ← Ê[a', S'].p'
  else p' ← p'_new s.t. D ⊨ R_S[Prob(a', a, p'_new, S')](S*)
  redoSit( N̂[do(a', S')], T̂(S_k))
```

Figure 3: algorithm for propagating the effects of discrepancies

Figure 3 shows functions that implement the patching, using the node and edge annotation as defined above.

[3]We can actually limit our attention to the list of fluents that actually occur in some annotated condition.

$\mathcal{R}_S[\varphi(do(\vec{a}, S))](S^*)$ denotes for the formula resulting from regressing formula φ over the action sequence $\vec{a}$, the instantiation in situation S^*, that is, substituting S^* for S. The call redoSit$(\hat{N}[S_k], \hat{T}(S_k), S^*)$ patches the tree $\hat{T}(S_k)$ from S_k to situation S^*, possibly making a different action the best (greedy) choice, in which case the annotation needs to be redone.

Proposition 1. By construction of the algorithm we have that after calling redoSit$(\hat{N}[S_k], \hat{T}(S_k), S^*)$, all annotated costs, rewards, probabilities, and heuristic values are with respect to S^* instead of S_k, i.e. in all situation labeled nodes $\hat{N}[do(\vec{a'}, S_k)]$, r' is s.t. $\mathcal{D} \models Reward(r', do(\vec{a'}, S^*))$ (or v' is s.t. $\mathcal{D} \models \bar{V}(v', do(\vec{a'}, S^*))$ in leaf nodes). In action labeled nodes $\hat{N}[a, do(\vec{a'}, S_k)]$, x is s.t. $\mathcal{D} \models (Poss(a, do(\vec{a'}, S^*)) \wedge x = 1) \vee (\neg Poss(a, do(\vec{a'}, S^*)) \wedge x = 0)$. And in edges $\hat{E}[a'', do(\vec{a'}, S_k)]$, c', p' are such that $\mathcal{D} \models \mathcal{R}_S[Cost(a'', c', do(\vec{a'}, S^*))] \wedge \mathcal{R}_S[Prob(a'', a, p', do(\vec{a'}, S^*))]$.

3.3 An Illustrative Example

Consider the following simplified example from the TPP domain where we modified the drive action to be stochastic. In this domain an agent drives from Depot to various markets to purchase goods. For simplicity, assume there is only one kind of good, two markets, and the following fluents: in situation s, $At(l, s)$ denotes the current location l, $Request(q, s)$ represents the number q requested of the good, $Price(p, m, s)$ denotes the price p of the good on market m, and $DriveCost(c, src, dest, s)$ the cost c normally incurred by driving from src to $dest$. Let there be two actions: $drive(dest)$ moves the agent from the current location to $dest$, and $buyAllNeeded$ purchases the requested number of goods at the current (market) location. The drive action is stochastic and may result in one of two outcomes $drive10(dest)$, and $drive12(dest)$, where the only difference is that the latter incurs a cost of 1.2 times the drive cost specified by $DriveCost(c, src, dest, s)$, whereas the former only incurs the normal cost (factor 1.0). This could represent the risk of a traffic jam on the route. Assume the planner has determined the plan $\vec{\alpha} = [drive(Market1), buyAllNeeded]$ to be optimal, but has as well considered $\vec{\beta} = [drive(Market2), buyAllNeeded]$ as one alternative among others. Note that we here simplified the representation of the plan by collapsing the two possible outcomes of drive actions into one straight line plan, as opposed to a conditional plan where the conditions are over the possible outcomes of the drive actions. This is possible, because the plan-suffixes for the two outcomes are identical in our example. For simplicity we also assume no rewards, i.e. $\mathit{Reward}(0, s) \equiv true$. The search tree resulting from this problem is shown in Figure 4. For the first step of the policy, we annotate the search tree $T(S_0)$ as follows, where for parsimony we ignore preconditions, rewards, and the heuristic value function used for evaluating leaf nodes.

$$\hat{E}[drive10(Market1), S'] = \\ (((\exists l).At(l, S') \wedge DriveCost(c, l, Market1, S'), 381), \\ (\mathit{Prob}(drive10(Market1), drive(Market1), p, S'), 0.8))$$

$$\hat{E}[drive12(Market1), S'] = \\ (((\exists c', l).At(l, S') \wedge DriveCost(c, l, Market1, S') \wedge \\ c = c' \cdot 1.2,\ 457.2), \\ (\mathit{Prob}(drive12(Market1), drive(Market1), p, S'), 0.2))$$

$$\hat{E}[drive10(Market2), S'] = \\ (((\exists l).At(l, S') \wedge DriveCost(c, l, Market2, S'), 458), \\ (\mathit{Prob}(drive10(Market2), drive(Market2), p, S'), 0.8))$$

$$\hat{E}[drive12(Market2), S'] = \\ (((\exists c', l).At(l, S') \wedge DriveCost(c, l, Market2, S') \wedge \\ c = c' \cdot 1.2,\ 549.6), \\ (\mathit{Prob}(drive12(Market2), drive(Market2), p, S'), 0.2))$$

$$\hat{E}[buyAllNeededSuccess, do(drive10(Market1), S')] = \\ (((\exists p', q').Price(p', Market1, S') \wedge \\ Requested(q', S') \wedge p = p' * q',\ 17), (true, 1.0))$$

$$\hat{E}[buyAllNeededSuccess, do(drive12(Market1), S')] = \\ (((\exists p', q').Price(p', Market1, S') \wedge \\ Requested(q', S') \wedge p = p' * q',\ 17), (true, 1.0))$$

$$\hat{E}[buyAllNeededSuccess, do(drive10(Market2), S')] = \\ (((\exists p', q').Price(p', Market2, S') \wedge \\ Requested(q', S') \wedge p = p' * q',\ 14), (true, 1.0))$$

$$\hat{E}[buyAllNeededSuccess, do(drive12(Market2), S')] = \\ (((\exists p', q').Price(p', Market2, S') \wedge \\ Requested(q', S') \wedge p = p' * q',\ 14), (true, 1.0))$$

Let's assume that even before we begin the execution of the plan, a discrepancy in form of an exogenous action e happens, putting us in situation $S^* = do(e, S_0)$ instead of S_0. How does this affect the relevant values in the search tree and, as a consequence, the optimal policy? This clearly depends on the effects of e. If e does not affect any of the fluents occurring in above annotated formulae, it can be ignored, the plan is guaranteed to remain optimal. This would, for instance, be the case when e represents the event of a price change on a market not considered, as that price would not find its way into the regressed formulae, which only mention relevant fluents.

Consider, for instance, the case where e represents the event of an increased demand, that is, increasing the value q of $Request(q)$, formally $\mathcal{D} \models Request(q, S^*) > Request(q, S_0)$. Then we need to reevaluate all conditions that mention this fluent, that is, in our example, the costs of all occurrences of the $buyAllNeededSuccess$ action. Afterwards any value that has changed need to be propagated up the tree, in order to determine whether the optimal policy has changed.

As a second example, imagine that the event e represents an update on the traffic situation between Depot and Market1, stating that the risk of a traffic jam has increased to 0.5 ($\mathcal{D} \models \mathit{Prob}(drive12(Market1), drive(Market), 0.5, S')$, $\mathcal{D} \models \mathit{Prob}(drive10(Market1), drive(Market), 0.5, S')$). Then we only need to recompute the backup values for the upper two branches without reevaluating any predicates except for these probabilities. After the backup, the optimal policy can again be greedily read off the tree.

4 Analysis

Three questions come to mind when trying to evaluate our approach. First, is the annotation of manageable size and does it scale? Second, what kind of guarantees can we make

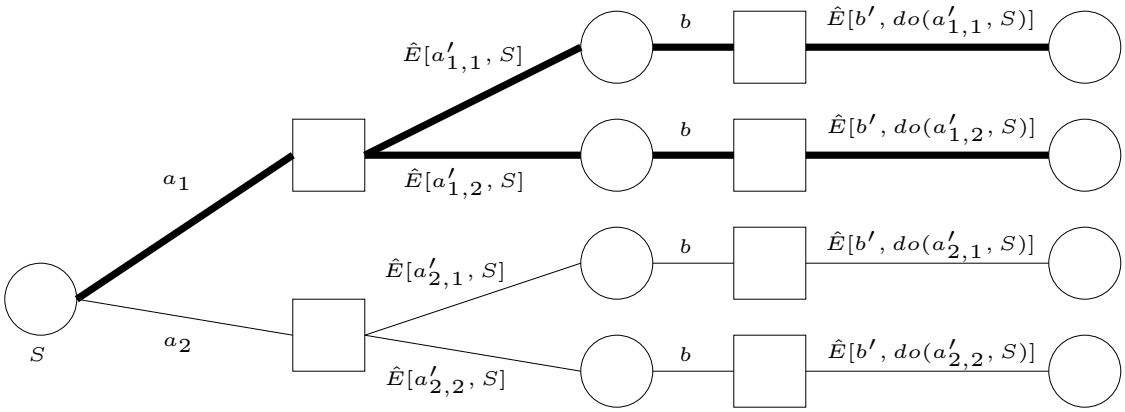


Figure 4: The annotated search tree of the example. Circles denote states, boxes denote nature's choices. The policy is marked by bold lines. For readability we abbreviate: $a_1 = drive(Market1), a_2 = drive(Market2), a'_{1,1} = drive10(Market1), a'_{1,2} = drive12(Market1)$, $a'_{2,1} = drive10(Market2), a'_{2,2} = drive12(Market2), b = buyAllNeeded, b' = buyAllNeededSuccess$.

about the quality of the value function (approximation) derived from the resurrected search tree? Last but not least, how does the approach perform compared to simple replanning from scratch, does it offer computational savings? We here address these questions in this order.

4.1 Space Complexity

The space complexity of our approach is determined by the space required to store the annotation. This looks dramatic at first, as we annotate a copy of the (sub-)search tree with each action of the policy. Let there be n actions ($n > 1$), let m be the maximal number of outcomes an action has, and let h be the maximal depth of the search tree. In the worst case, the tree is uniformly expanded to depth h, each action is possible in each situation-node, and each action has m outcomes. Then the search tree has size $(nm)^h$. It turns out, that the annotation is not significantly larger and is in fact only constantly so in h, despite the number of sub-tree copies at each subsequent level of the policy.

Theorem 1 (Annotation Size)**.** If the search tree has size $X = (nm)^h$, then the size of the annotation is in $\mathcal{O}(\lambda X)$ with $\lambda = \frac{n - \frac{1}{n^h}}{n-1}$.

Proof: Recall the format of the annotated policy from Section 3.1. In the root, $T(S)$ has size $c(nm)^h$ for some constant c reflecting the per node annotation. We annotate each of the (at most) m outcomes a' of a with a copy of the resp. sub-search tree of the remaining horizon $h-1$, in total $cm(nm)^{h-1}$. This goes on for h steps, eventually annotating m^h leaves with a constant size value. In total we annotate

$$\begin{aligned} & c(nm)^h + mc(nm)^{h-1} + \cdots + m^h c(nm)^{h-h} \\ = \quad & c(nm)^h(1 + \frac{1}{n} + \cdots + \frac{1}{n^h}) = c(nm)^h \frac{n - \frac{1}{n^h}}{n-1} \end{aligned} \qquad \square$$

Because $\lim_{n\to\infty} \lambda = 1$, this means that we generally have no space requirements above those required to store the search tree in the first place.

4.2 Optimality Considerations

What can we claim about the quality of the resurrected value function we obtain after patching a tree from situation S to S^*? This depends on the usage, i.e. the problem being solved and the type of planner used. In this section we do the analysis for two classes of MDPs.

Finite Horizon MDPs

Assume the task is to solve an MDP of finite horizon H starting in situation S_0, given an admissible heuristic function[4], and that the planner has expanded the tree to a maximal depth of H, possibly pruning some branches according to standard pruning techniques, as in A^* search. The value according to the thus obtained value function for any situation S at level $2h$ of the tree is precisely $V^*_{H-h}(S)$ and the maximizing actions describe the optimal policy. Our approach can be used to verify the continued optimality of this policy in the face of discrepancies.

Theorem 2. A policy $\pi(S)$ starting from situation S and annotated with $\hat{T}(S)$ as in Definition 1 continues to be optimal in S^* if after calling redoSit$(\hat{N}[S], \hat{T}(S), S^*)$, the greedy policy for $\hat{T}(S)$ coincides with $\pi(S)$.

Proof Sketch: The theorem follows from the admissibility of the heuristic function and Proposition 1.

Sampling in Large MDPs

In [4], Kearns et al. show that the time required to compute a near-optimal action from any particular state in an infinite horizon MDP with discounted rewards does not depend on the size of the state space. This is significant, as it allows, at least theoretically, for the computation of near-optimal actions even in infinite state MDPs. They propose a sampling based decision tree search algorithm **A** and prove bounds on how many samples C and what horizon H is necessary

[4] An admissible heuristic function is an estimate of the real value function that doesn't underestimate the real value for any state.

in order to obtain an ϵ-optimal action. The algorithm works like the one described in Section 2.2, but only explores C outcomes from the set of outcomes of each action, where the samples are chosen according to the probability distribution over the outcomes. The bounds C and H only depend on ϵ, the discounting factor γ, and a bound R_{max} on the absolute value of the reward function. The bound on the horizon is $H = \lceil \log_\gamma(\epsilon(1-\gamma)^3/(4R_{max})) \rceil$. The algorithm runs in the initial situation S, returning a best action a. After executing a and observing the outcome a'_i, the algorithm needs to be re-run in $do(a'_i, S)$, because (i) the error-bound of the value function approximation for node $do(a'_j, S)$ for any outcome a'_j of a is greater than ϵ, but more severely (ii) it is unlikely that the actual action outcome a'_i is among the C samples considered in planning. The latter is particularly true in *continuous* domains with continuous sets of action outcomes where this probability becomes 0.

We argue that the former issue is not severe, and that the latter can in many cases be accounted for using our approach. Simple term manipulation of the above horizon bound yields the error-bound for considered successor situation $do(a'_j, S)$ of the initial situation as $\epsilon' < 4\gamma^{H-1}R_{max}/(1-\gamma)^3$. This error bound on the value function can be resurrected by applying our algorithm with the limitation that neither action outcome probabilities nor preconditions must have been affected by the discrepancy.

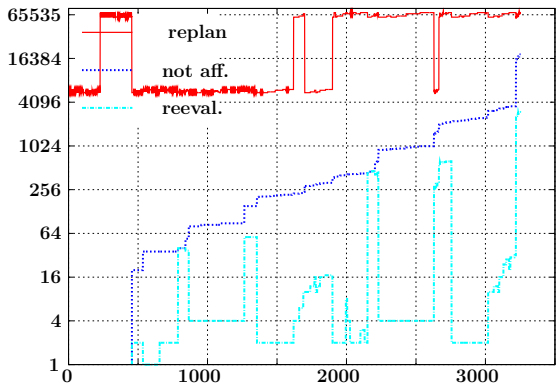
Theorem 3. Let T be the search tree as spanned by the algorithm **A** [4] with horizon H and sample width C from initial situation S, $\pi(S)$ be the greedy policy extracted from T with best-first action a, and $\hat{T}(S)$ be the annotated search tree (cf. Def. 1). Let the execution of a yield the actual situation S^*. If there is a node $\hat{N}[do(a'_j, S)]$ in $\hat{T}(S)$ such that $\mathcal{D} \models \varphi(do(a'_j, S)) \equiv \varphi(S^*)$ for all φ s.t. φ is a regressed precondition or regressed probability predicate in the annotated search tree $\hat{T}(S)$, then, after calling redoSit$(\hat{N}[do(a'_j, S)], \hat{T}(do(a'_j, S)), S^*)$ the value function V' described by $\hat{T}(S)$ is such that $|V'(S^*) - V^*(S^*)| < 4\gamma^{H-1}R_{max}/(1-\gamma)^3$.

This is particularly useful when gauging the relevance of exogenous events. If situation S is expected but the actual situation $do(e, S)$ for some exogenous event e is observed, then, if e doesn't affect any preconditions or action probabilities, redoSit$(\hat{N}[S], \hat{T}(S), do(e, S))$ resurrects the approximation quality of the current policy for the new situation.

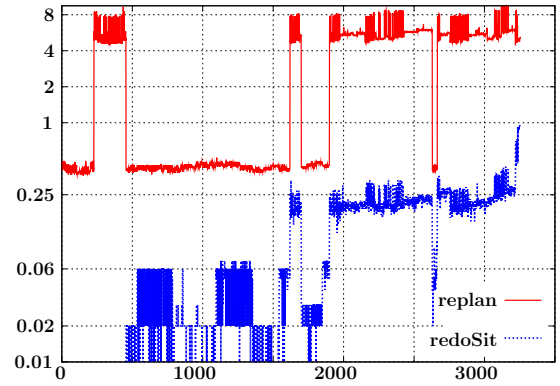
4.3 Experiments

We were interested in determining whether the approach was time-effective – whether the discrepancy-based incremental reevaluation of the search tree could indeed be done more quickly than simply replanning when a discrepancy was detected. The intuition was that most discrepancies only affect a small subset of all fluents and that this effect often doesn't propagate to relevant values. In this section we show the results of some experiments we did which support this intuition practically.

To this end, we compared a preliminary implementation of our `redoSit` algorithm to replanning from scratch on



(a) # of conditions



(b) running time in seconds

Figure 5: replanning vs. tree patching (using logarithmic scale)

different problems in a variant of the metric `TPP` domain of the 5^{th} International Planning Competition, where we changed some actions to have stochastic outcomes. In each case, we uniformly spanned the search tree up to a particular horizon, perturbed the state of the world by changing some fluent, and then ran both `redoSit` and replanning from scratch. To maximize objectivity, the perturbations were done systematically by multiplying the value of one of the numeric fluents (driving costs between locations, prices and number of goods on sale on different markets, number of requested goods) by a factor between 0.5 and 1.5 (step-size 0.1) or by redistributing 0.5 of the probability mass of some action's outcomes. In total we tested 3256 cases. Figure 5 shows from top to bottom: the number of relevant conditions as evaluated by replanning, the number of actually affected conditions, and the number of such conditions that are unique. The latter is the number of conditions that actually need to be reevaluated. Values for all other affected conditions, effectively copies of earlier seen ones, can be obtained from a cache of evaluated conditions. The number of unique affected conditions is extremely low. On average this number was 9751.03 times lower than the number of relevant conditions. This strongly motivates a patching approach over replanning. To enhance readability we ordered the test cases by the number of affected conditions.

5 Related Work

Our idea of annotating a policy with the relevant conditions for its quality is inspired by earlier work in classical planning. The idea has been exploited in a number of systems to monitor the continued validity of a straight-line plan during execution (e.g., [6], also cf. [3]). None of these system, however considered more complex goals or stochastic actions.

Except for our earlier work [3], to the best of our knowledge the SHERPA system [8] is the sole previous work that addresses the problem of monitoring the continued optimality of a (deterministic!) plan. However, this system, which is a lifted version of the Life-Long Planning A^* (LPA^*) search algorithm, is limited to discrepancies in the costs of actions.

Also related is [11] in which the authors exploit the 'rationale', the reasons for choices made during planning, to deal with discrepancies that occur during (partial-order) *planning*. The authors acknowledge the possibility that previously sub-optimal alternatives become better than the current plan candidate as the world evolves while planning, but the treatment of optimality is informal and limited.

Other methods for creating robust policies under time constraints include [10]. The authors consider finite horizon MDPs with an explicit goal area and assume there is a method for creating some path with positive probability from the initial state to the goal. They propose to then, time permitting, extend this path to an *envelope* of states by successively including additional states that are on the fringe of the current envelope, i.e. possible successors of certain actions when executed in a state inside the envelope. This way, successively more contingencies are added to the policy. This approach differs from ours, as it relies on knowledge about an accurate model of the world and thus does not increase robustness against completely unforeseen courses of events.

Also Real-Time Dynamic Programming [1] (RTDP) addresses the problem of incrementally creating a policy for the accessible region of state space given an initial state. RTDP works by interleaving value function estimate improvement and execution. The action executed is always the greedy one according to the current estimate. Interleaved with execution, some form of look-ahead search from the current state is performed and the values of visited states backed-up accordingly. As with Dean et al.'s approach, also RTDP relies on an accurate model of the world.

Strongly related to our approach is the work on solving first-order MDPs by Boutilier et al. [2] and more recently Sanner et al. [9]. This work proposes first-order decision-theoretic regression (FODTR) to solve First-Order MDPs exactly for all states, as opposed to a particular initial state. The approach works, roughly, by repeatedly regressing the value function and rewards, both represented as first-order formulae of a particular form, over stochastic actions, providing an improved value function for abstract states, where the abstraction is induced by the regression. There are certain limitations to this approach, in particular it does not generally allow for continuous domains. In real-world systems it further seems beneficial to follow a forward search approach to focus on the reachable subset of state space. As such our approach explores a middle-ground between this and plain decision-tree forward search.

6 Summary and Future Work

We have presented an algorithm that lifts an idea commonly used to monitor the validity of deterministic plans to decision theoretic planning in relational MDPs, monitoring the adequacy and potential optimality of a policy. The approach works by annotating the policy with conditions regressed to the situations where they are relevant. In so doing, the discrepancy between an unplanned-for state and an expected state can be evaluated with respect to the objective of a near-optimal or optimal policy. The intuition that this can be significantly more efficient than replanning, should something unexpected happen, is supported by experiments with a preliminary implementation. In future work we intend to perform further theoretical analysis of our approach, and to run more experiments, in particular in continuous domains.

References

[1] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[2] C.Boutilier, R.Reiter, and B.Price. Symbolic dynamic programming for first-order MDPs. In *Proc. IJCAI'01*, 690–700, 2001.

[3] C.Fritz and S.McIlraith. Monitoring plan optimality during execution. In *Proc. ICAPS-07*, 2007. (to appear).

[4] M.Kearns, Y.Mansour, and A.Y.Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *Proc. IJCAI-99*, 1324–1231, 1999.

[5] R.Dearden and C.Boutilier. Integrating planning and execution in stochastic domains. In *Proc. AAAI Spring Symposium on Decision Theoretic Planning*, 55–61, 1994.

[6] R.Fikes, P.Hart, and N.Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[7] R.Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 2001.

[8] S.Koenig, D.Furcy, and C.Bauer. Heuristic search-based replanning. In *Proc. AIPS'02*, 294–301, 2002.

[9] S.Sanner and C.Boutilier. Approximate linear programming for first-order MDPs. In *Proc. UAI05*, 509–517, 2005.

[10] T.Dean, L.Kaelbling, J.Kirman, and A.Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.

[11] M.M. Veloso, M.E. Pollack, and M.T. Cox. Rationale-based monitoring for continuous planning in dynamic environments. In *Proc. AIPS'98*, 171–179, 1998.