Policy Regression for Monitoring Execution in Goal Reasoning Systems

Noah Reifsnyder NDR217@LEHIGH.EDU
Hector Munoz-Avila HEM4@LEHIGH.EDU
Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA 18015 USA

Abstract

In this paper we investigate the problem of representing and defining policy regression expectations, needed for execution monitoring, when the course of action being monitored is represented as a policy. Until now, the notion of expectation for goal reasoning systems has focused on plans, defined as sequences of actions. This paper presents a formalization to the notion of expectations when the agent is executing policies instead of plans. It also relaxes the notion of policy. Policies are used when acting in probabilistic domains, where executing actions might have multiple outcomes according to some probability distributions.

1. Introduction

There has been an increasing interest in AI Safety, the creation of reliable AI agents that don't step out of their boundaries. Part of the interest on AI Safety is the realization that as systems increase in sophistication they may behave in ways that are inconsistent towards the agent's own specifications as encoded in its plan formulation knowledge. Planning knowledge can be formulated as a collection of actions, $\mathcal{A}$, indicating how states change when the actions are applied. Actions may have deterministic outcomes indicating a pre-determined outcome or non-deterministic indicating multiple possible outcomes; the latter requires planning paradigms that plan ahead for each possible outcome.

A related theme in AI research are planning paradigms providing provable guarantees that a generated solution π solves a given problem $\mathcal{P}$. Such guarantees can be seeing as providing a component of AI's answer to the problem of AI Safety; namely, *as long as the solution π is followed, and no unaccounted contingency occurs, we guarantee that the problem will be solved as specified in $\mathcal{P}$.*

Another component of the answer from AI to the problem of AI Safety is the monitoring of π's execution. There are two problems that must be addressed for this component: **Detection**: determining if the agent is deviating from the course of action plotted in π; **Correction**: taking corrective action when such a deviation is detected. There are multiple approaches to address the correction problem including (1) replanning, namely, generating a new solution from the current state s reached where the failure was detected (Fox et al., 2006; Warfield et al., 2007); (2) goal reasoning, where the agent performs a meta-reasoning process and formulates a new goal (Aha, 2018;

Cox, Dannenhauer, and Kondrakunta, 2017); (3) or simply the agent stopping its own execution (e.g., in Gregg-Smith and Mayol-Cuevas (2015) a robotic arm deactivates itself when encountering a discrepancy).

In this paper we are focusing on the detection problem, when the solution π is a policy. Policies are generated by underlying assumptions that the agent knows all outcomes and their probability distribution (e.g., ND planning) or by learning from interactions with the environment (e.g., Reinforcement Learning). These policies fail the moment a situation is encountered that was not pre-planned for in advance. This has become an issue of increasing concern in reinforcement learning (and AI in general), as the agent may take too many cycles before adapting with potentially catastrophic consequences (Mason et al., 2017).

Expectations allows agents to detect such unexpected events occurring in the environment (Cox, 2007; Dannenhauer, 2018) . They allow the agent to know not just what went wrong that caused the policy to fail, but also allow the agent to ignore unexpected events that don't actually affect the achievement of the goals. For example, in the Blocks World, if a new block appears, it would cause an expectation's failure. There will be no mappings from a state with an added block to an action. With expectations, we will be able to determine if this block will hinder the current policy or not and in the latter case simply continue with the policy's execution. This allows the agent to ignore random variations of the state that don't impact the achievement of the goals.

We re-examine the notion of expectations where solutions achieving the goals are policies, which are mappings from states to actions, $\pi : S_\pi \rightarrow A$. Policies are used in probabilistic domains, where actions may have multiple possible outcomes. A policy accounts for all foreseen states $s \in S_\pi$ that the agent might encounter. However, agents might encounter situations it has not planned for; that is $s \notin S_\pi$. This can happen in particular when the agent is operating autonomously for extended periods of time. Reasons for encountering such unforeseen situations include: (1) exogenous events, that is, conditions in the state that were not planned for and (2) changes in the actions. An example of the latter is a failure in a mechanical motion device that causes it to operate in a different way from what is modeled in the action definitions. Cox and Ram (1999) present a taxonomy of failure reasons that is attributable to factors such as discrepancies in the state, discrepancies in the action model, discrepancies in the goals and discrepancies in the environment.

The following are the main contributions of this paper: (1) A procedure for computing goal regression for policies; (2) A relaxation of the notion of policy, allowing execution even when the state reached doesn't match any of the states $s \in S_\pi$ defined in the policy π; (3) Properties of our procedure; and (4) An empirical study of policy regression expectations in two variants of domains from the goal reasoning literature, Arsonist (Paisner et al., 2013) and Marsworld (Molineaux, Kuter, and Klenk, 2012).

2. The Notion of Expectations and Their Representation

To address the detection problem, agents compute the expectation, $X_{(\pi,s)}$, as a function of the planning problem's solution π and the current state s (Dannenhauer and Munoz-Avila, 2015). The agent checks if this expectation is met in s; that is, if $X_{(\pi,s)} \subset s$. Expectations are conditions that are checked against the observed state s during execution. When the conditions are not met, a discrepancy occurs that must be addressed (Munoz-Avila, Dannenhauer, and Reifsnyder, 2019). Expectations are needed, because the environments are dynamic and the conditions under which a plan was generated may change at execution time (Cox, 2007). A technical challenge we are facing in our work is that we are considering the case when π are policies, $\pi : S \rightarrow A$. As a result, if $ND(a, s)$ are the possible states that can be reached after executing a on state s, it is possible that the observed state $s' \notin ND(a, s)$ and hence continuing execution of π is no longer possible.

The problem of computing the agent's expectations for π is deceivingly simple; at first glance it would seem sufficient to check that the preconditions of a are satisfied in s (i.e., to define $X_{(\pi,s)}$= *"the preconditions of the next action to execute,* $\pi(s)$*"*) but this would result in a myopic agent that is not checking the plan trajectory in π; alternatively, we could project s_0 based on π to the state s right when a is to be executed (i.e., to define $X_{(\pi,s)} = s$). This is how expectations are frequently defined in goal reasoning systems (Aha, 2018; Muñoz-Avila et al., 2010; Molineaux, Klenk, and Aha, 2010). The problem is that any change in the expected state will trigger a correction step even if the discrepancy is unrelated to π's trajectory. Researchers have observed that the notion of expectations plays a key role in the resulting performance of goal reasoning agents (e.g., Dannenhauer and Munoz-Avila (2015)). If the expectations are too narrow, agents might fail to detect discrepancies when they are needed leading an agent to further commit to actions derived from π that will lead to a failure. If the expectations are too general, the agent will flag a failure which will trigger a process (e.g., replanning) to address the failure when it is not required to do so.

The questions that we propose to address in this paper are the following:

- When monitoring the execution of a policy π, how to compute the regression expectations, $X_{(\pi,s)}$, of the current state s?
- If a discrepancy is detected during monitoring execution, is it possible to continue executing π?

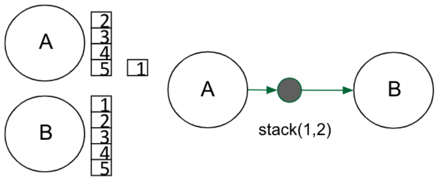

Figure 1: Two states, A and B, showing the final action in a Blocks World domain where the goal is to have 5 blocks stacked on top of each other.

Computing Regression Expectations for plans (i.e., sequences of actions) is well understood (Pollock, 1998); we give an example in the Blocks World domain. Consider states A and B as shown in Fig. 1, where state B is the goal state, and A is one action away. The action $stack(1, 2)$ places block 1 on block 2, achieving the goal $on(1, 2)$ while the remaining goals remain intact, $on(2, 3), on(3, 4), on(4, 5)$. The Regression Expectations, $X_{(\pi,B)} = \mathcal{G}$; since B is a goal state then the expectations are the goals themselves. To find the Regression Expectations for state A, we start with the Expectations for state B, subtract out the effects from the action between them, $on(1, 2)$ and add in the preconditions of the action $stack(1, 2)$ (i.e., Block1 and Block2 have no blocks on top of them). Thus, the Regression Expectations of state A is: $X_{(\pi,A)} = \{on(2, 3), on(3, 4), on(4, 5), clear(1), clear(2)\}$. Using the Regression Expectations for state A as a model, we can understand the importance of having this set of expectations. Regardless of any other variations in the state, we can see that if these values hold true, then the plan will satisfy the goal. The preconditions of the final action are met, and once we add the effects of the action to the rest of the expectation set, the goal is satisfied.

3. A Sample Domain

We illustrate our work with the Arsonist World (Paisner et al., 2013). Similar to blocks world, the goal is to allocate blocks to form a desired configuration. For instance, the agent might want to form a single tower of n blocks. Unlike blocks world, there is an arsonist that ignites blocks at random. There are only two actions in this domain: $stack$ and $unstack$. $unstack(?b1, ?b2)$ is a deterministic action that takes a block $?b1$ on top of some block $?b2$ with no block above it and places $?b1$ on the table (i.e., with the assignment *below(?b1)* ← *none*); $stack(?b1, ?b2)$ requires that no blocks are on top of $?b1$ and $?b2$ and has three probabilistic outcomes; either block $?b1$ is placed on top of block $?b2$, block $?b1$ falls to the floor (i.e., with the assignment *floor(?b1)* ← *True)*, or block $?b2$ is knocked off the tower (i.e, both block $?b2$ and block $?b1$ end up on the table). Stacking the block correctly occurs 90% of the time, while knocking the block off the tower occurs 8% of the time and knocking the block on the floor occurs 2% of the time. $stack$ and $unstack$ are shown in Table 1, which use the state-variable representation (Ghallab, Nau, and Traverso, 2004) (Section 2.5): the variable *onfire(?b)* returns true iff $?b$ is on fire. The variable *above(?b)* returns the block immediately above $?b$ and *none* if no block is above $?b$. The variable *below(?b)* returns the block immediately below $?b$ and *none* if $?b$ is on the table.

Consider an example, when the initial state starts with 5 blocks (see Table 2): 1, 2, 3, 4, and 5. We have a goal to create a tower of 5 blocks: 1 on top of 2, 2 on top of 3, 3 on top of 4, and 4 on top of 5.

Figure 2 shows an example of a **policy** solving this problem. For every possible state, $s_0, \ldots s_4$ the agent can find itself in, it indicates the action it should take (in this case stack), and the possible states it will reach based on the actions' probabilistic outcomes.

4. Preliminaries

```
(:operator stack
:parameters ?b1 ?b2
:condition above(?b1)=none, above(?b2)=none,
onfire(?b1)=False
:effect(.9) above(?b2)← ?b1, below(?b1)←?b2
:effect(.08) below(?b2) ← none
:effect(.02) floor(?b1)=True
```

```
(:operator unstack
:parameters ?b1 ?b2
:condition above(?b2)=?b1, above(?b1)=none
onfire(?b1)=False
:effect above(?b2) ← none, below(?b1) ← none )
```

Table 1: stack and unstack operators. Probabilities associate with effects are listed in parenthesis at the beginning of the effect definition

```
(:Initial State
{onfire : {1: False, 2: False, 3: False, 4: False, 5: False}}
{floor : {1: False, 2: False, 3: False, 4: False, 5: False}}
{above : {1: none, 2: none, 3: none, 4: none, 5: none}}
{below : {1: none, 2: none, 3: none, 4: none, 5: none}}
:Actions
stack, unstack
:Goals
{above : {2: 1, 3: 2, 4: 3, 5: 4}}
```

Table 2: Planning problem, where there are 5 blocks enumerated 1 through 5. We use a compact representation for the state: we write "1:False", "2:False" in a table for onfire, instead of writing onfire(1) = False, onfire(2) = False, etc.

This section provides intuition into the definitions of the basic terminology we use. For the formal definitions please refer to Appendix 1. We represent the states as a collection of variables V, that each take some constant but mutable value. A state s indicates the binding of the value $s(v)$ of each variable $v \in V$. For instance, the variable s_0*(above(1))* returns the value *None* indicating no blocks on top of block 1 in the initial state.

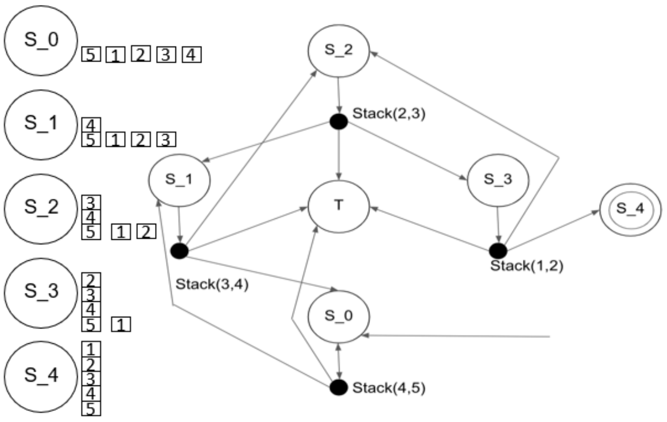

Figure 2: A policy for given state from Table 2 and operator definitions in Table 1. s_0 is the initial state, and s_4 is the terminal state that achieves the goals. T represents a terminal state at which the policy fails to achieve the goals. Actions are represented by the black dots.

As exemplified in Table 1, an operator is a 4-tuple *o=(name parameters precondition effect)*. The parameters are a collection of free variables, which are used to facilitate writing operators.[1] We denote free variables by using "?". For instance *?b1* denotes the free variable b1 (i.e., a block).

The preconditions are a list of variable value pairs that must be true in the state for the operator to be applicable. For example, the condition *above(?b1)=none* means there must be no block above block ?b1. All conditions listed in the operator must be true for the operator to be applicable.

The effects are a list of variable assignments that take place when the operator is applied to a state. For example, *above(?b2)* $\leftarrow$ *?b1* indicates that ?b1 will be above ?b2 after the operator is applied (i.e. *s(above(?b2))* will return ?b1 after the operator is applied).

[1] We call these free variables to distinguish them from the state variables.

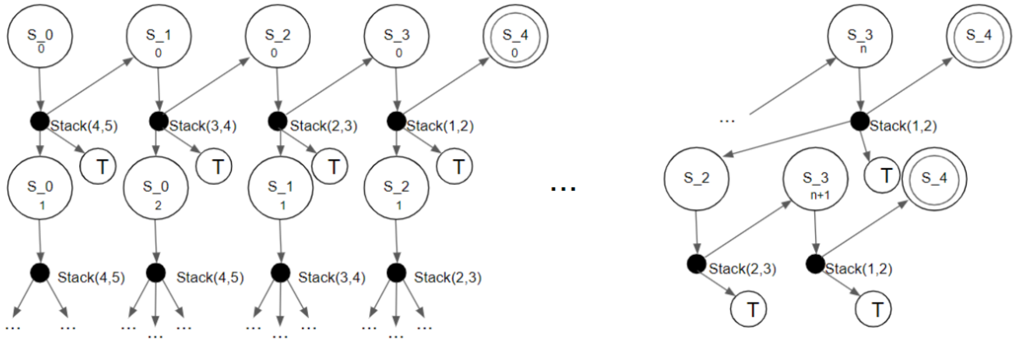

Figure 3: The expanded plan tree T_π used for calculating policy goal regression expectations. The numbers next to vertexes are the order that vertex is visited in our search.

The goals are a list of variable value pairs that must be true at the end of execution of solution π for π to be considered successful.

5. Regression Policy Expectations

Since we are regressing over a policy which is a graph, we need to do some pre-processing mainly to ensure no infinite loops are encountered during the regression. To define regression on a policy π, we perform two steps: (1) Compute the plan tree T_π derived from π; and (2) Compute regression on T_π as a proxy for computing regression on π.

Constructing T_π is done by making the starting state s_0 of π the root of T_π. Then starting with s_0, we recursively add all children to T_π from π: if the node is a state s in π, it has as a single child, the action $\pi(s)$. If the node is an action $\pi(s)$, then its children are the states in $ND(\pi(s), s)$. To handle loops, any time an edge that points back to a node that is an ancestor, that path is not expanded again during the construction of T_π in that family (i.e. only from future descendants is this edge excluded, it could be expanded on a different path). Figure 3 shows the tree T_π obtained from the policy π (Figure 2). For details on the construction of T_π, see Appendix 2.

If a state s occurs more than once in T_π, then we define the expectations for s to be the expectations for the first time s is listed in the topological sort of T_π (Cormen et al., 2001) (Section 22.4). Informally, it selects the one closest to the initial state, which is the one that includes all possible paths from the state to a terminal state. For instance, in Figure 3, s_0 occurs multiple times and the one selected is the root, s_{0_0}.

5.1 Computing Expectations

Informally, the agent computes the expectations for a state by regressing the goals and necessary conditions to reach them. This is done by having each node take the expectations of all its children modified by the probability of reaching each child, then adding in the preconditions of its own action. The expectations of the leaf nodes are either the goals with 100% probability if they are a goal state, or $\varnothing$ with 100% if it is a non goal terminal state. The $\varnothing$ expectation denotes a failed execution. This means each set of expectations is a list of variable value pairs, with an associated probability. The probability is thus representative of how important that specific variable value pair is to the overall success of the agent. For example, in Figure 4 the two children states B and C have two different variable bindings for v (b and c respectively). When regressed back to the parent state A, their probabilities are cut in half to represent the probability that specific binding is needed. Then the preconditions for A's action are added in, and we are left with the final expectation set for state A. Appendix 3 provides all the formal details.

5.2 Properties

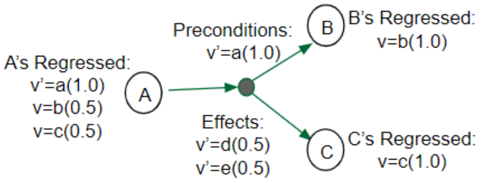

Figure 4: An Example of Regressing a set of expectations from 2 children nodes to the parent.

In Appendix 4 we formally state and prove the following properties: (1) Our procedure will result in a unique T_π for any given policy π. (2) The size of T_π is polynomial on the size of the policy (i.e., number of states plus number of state-action transitions). (3) T_π subsumes all other trees T' derived from π. A tree T' is derived from π by expanding on the states and actions in the same way as when constructing T_π , however depending on the order of nodes expanded, you can end up with a tree T' whose structure is different. The subsumption property states that for every complete path P' in T', if we compute the expectation for the first occurrence of any state s in P', $X_{(T',s)}$, then there is a complete path P in T_π such that for the first occurrence of state s in P, the expectations for T_π, T', and π are the same.

Theorem 1 guarantees that expectations are well defined since they are generated from a unique plan tree T_π. Theorem 2 implies that the procedure constructing T_π runs in polynomial time on the size of $G(\pi)$. Theorem 3 implies that goal regression on T_π accurately calculates goal regression over π since T_π subsumes all possible complete paths in π

6. Monitoring Policy Execution

For Policy Regression we need to take into account the probability distribution of the values of a variable. To do so, we detect discrepancies in an observed state s as follows: Let $X_{(\pi,s)}(v)$ be a function that represents the expectations for variable v. By definition, $X_{(\pi,s)}(v)$ is a probability distribution for all values that v may take. For example, in Figure 4, $X_{(\pi,A)}(v) = \{v = b(0.5), v = c(0.5)\}$. We add the probabilities of values that are not equal to value of v in the observed state, $s(v)$: $P = 1 - (\Sigma_{s(v) \neq c', (c',p) \in X_{(\pi,s)}(v)}(p) + Pr(X_{(\pi,s)}(\varnothing)))$. The term $Pr(X_{(\pi,s)}(\varnothing))$ is the probability we will end in a terminal non goal state from the current state s. Thus, P indicates the probability the agent will still succeed depending on the value of $s(v)$. A domain-specific parameter, δ, is needed to trigger a discrepancy. We say that a discrepancy occurs if $P < \delta$. In our experiments, we set $\delta = 0.5$ as a threshold as it denotes that the execution of the policy is more likely to fail than succeed.

Since policies choose an action based on the current state the agent is in, we need to relax the notion of policy by slightly modifying how an agent chooses its actions when executing a policy π. If we continue to base action decision off of the entire state, we render the expectations moot, because as soon as any deviation occurs the policy will fail; it will encounter a state s that is not defined in the policy. There are times that values in the state will change that don't affect the progress of the agent: when the expectations are met; since the expectations are a subset of s, it is possible to visit a state s' such that: $s \neq s'$ but the expectations are a subset of s' and therefore s' meets the expectations. To handle this, we use the expectation sets themselves to identify the states and choose actions with the policy.

Since we are computing the necessary conditions for the policy to succeed with a probability P, any state matching the expectation set will succeed with a probability P. Since we are weakening the notion of a state in a policy, namely, requiring only a subset of the state's atoms to be true, a few things can happen. It is possible for multiple sets of expectations to hold true at any given point in execution. We handle this in two steps: (1) Because we know the set of states $ND(a, s)$ the agent could be in after taking an action a from state s, we narrow our choices to that set. (2) If there is more than one state $s' \in ND(a, s)$ for which the expectations are satisfied, we choose the state where the probability P of succeeding is the highest.

When executing a policy π in an state s, the agent executes $\pi(s)$ and reaches a state s'. It is possible that this state s' doesn't match any of the successor states defined in the policy π (i.e., $s' \notin ND(\pi(s), s)$) . If the expectations $X_{\pi,s''}$ for any successor state $s'' \in ND(\pi(s), s)$ are met, then $\pi(s'')$ can be applied as in (1) and (2) above.

7. Empirical Evaluation

In our experiments, we compared 4 expectation types: Immediate, Informed, policy regression without goals (regressing beginning with an empty set) and regressing when the goals are known (in the figures we call it G-Regression).[2] Immediate and Informed are defined in Dannenhauer, Munoz-Avila, and Cox (2016); succinctly, Immediate expectations check the preconditions of the next action a to execute in the current state s and their effects. Because we are dealing with probabilistic domains, we modified them to check for the resulting state s', $s' \in ND(a, s)$ holds. Informed expectations accumulate forward the effects of all actions executed so far. In the case of probabilistic domains, each executed action commits to one of its ND effects. This effect is the one we accumulate forwards.

For planning, we implemented a probabilistic domain-configurable planner as described in Kuter and Nau (2005). The policies generated were solutions for the domain with probability ρ. Aside from the different expectations, the agent was the same: it has a simple module that given a

[2] A situation where goals are not known is when HTN planning techniques are used to generate the policies. In HTN planning tasks not goals are used to specify the problems Erol, Hendler, and Nau (1996). Tasks does not necessarily map to atoms in the state.

discrepancy, it always generates the goal based on a mapping from discrepancies to goals $d \rightarrow g$. The performance metric is the same as in Dannenhauer, Munoz-Avila, and Cox (2016): the cost of the executed plan until a terminal state is reached (see below for a description of the cost function). At that point we check if the goals are satisfied; if they are not the execution is considered a failure.

The first domain is a variant of the Arsonist domain (Paisner et al., 2013), which is itself a modified version of Blocks World. The goal is to make a tower of 10 blocks. There is an arsonist that is arbitrarily setting blocks on fire (i.e., a block on fire is an exogenous event). The cost of a problem is the accumulated number of actions where a block that ends in our tower is on fire (each block adds 1 point for every action taken while it's on fire. The possible actions for the agent in the Arsonist domain are the usual `stack` and `unstack`, both requiring the block not to be on fire. When there is a discrepancy because a block is on fire, the agent triggers a goal to remove the fire. Afterwards the agent continues achieving the goal. In our variant, actions have probabilistic effects: stacking a block on top of another block has the same probabilistic outcomes as in Table 1: stacking the block correctly occurs 90% of the time, while knocking the block off the tower occurs 8% of the time and knocking the block on the floor occurs 2% of the time. A failure occurs if any block in the tower is on fire or if the tower is not 10 blocks tall when the agent finishes executing actions. Our planner created a policy successful with probability $\rho = .83$

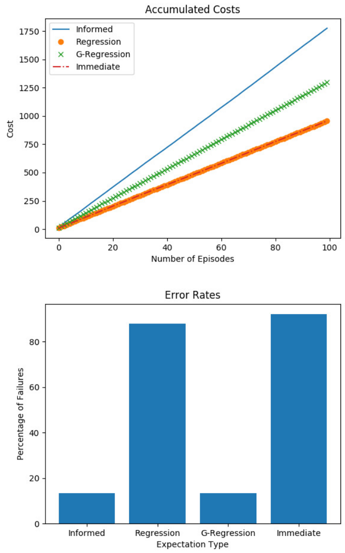

Figure 5: (a) Accumulated costs - Arsonist Domain; (b) Failure rates - Arsonist Domain

Figure 5(a) compares accumulated costs between Immediate, Informed, Regression and G-Regression. Informed Expectations had the highest cost. This is because blocks were allowed to burn until they were taken to stack onto the tower; the agent using Informed Expectations had no way of knowing which blocks it would eventually use. Policy Regression and Immediate Expectations performed similarly to one another. They have lower costs because their expectations have no knowledge of previously stacked blocks, and thus do not know if the last `stack` action knocked a block off the tower. Therefore they both perform the 10 `stack` actions necessary and then terminate, regardless of if the tower holds 10 blocks or not. Policy Regression and Immediate Expectations had near 100% failure rates (as seen in Figure 5(b)), due to the same lack of knowledge in the expectations. (i.e., at some point a block is knocked off and was never re-stacked, or a final block was on fire). G-Regression had a cost well below informed Expectations, and a failure rate of 17%. This is because it infers the knowledge of which blocks it would eventually use in the tower (i.e., by knowing which blocks would be in the final tower from the goals) and could put the fires out immediately. The 17% failure rate is the inherent probability of ending in a non goal terminal state based on the probabilistic outcomes of the actions (i.e., a block was knocked to the floor and thus the tower could never be fully built).

The second domain is a probabilistic version of Marsworld (Dannenhauer, Munoz-Avila, and Cox, 2016), which is itself a variant of Mudsworld presented in Molineaux and Aha (2014). In this domain the agent navigates a $N \times N$ grid seeking to turn on 3 randomly placed beacons (we use $N = 10$). The agent can take actions to move `forward`, `backward`, `up`, or `down`. We added probabilistic outcomes: each movement action had a .9% chance to move successfully, or a .1% chance to take the counter-clockwise direction (e.g., the action `forward` could result in moving up). The agent also has a finite fuel resource that is required by and used by the actions. Once the agent runs out of fuel, we say the agent has reached a non-goal terminal state (i.e., a "dead end"). For this domain, our planner created a successful policy with probability $\rho = .81$

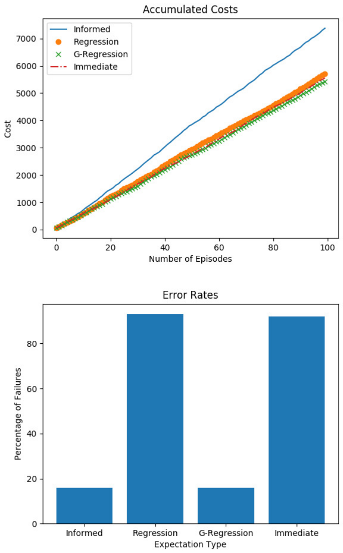

Figure 6: (a) Accumulated costs - Marsworld Domain; (b) Failure rates - Marsworld Domain

Like in Dannenhauer and Munoz-Avila (2015), there are two exogenous events that were not planned for: first, the agent might get stuck in mud, requiring replanning to get unstuck. Once unstuck, the agent continues seeking to turn on the beacons. Each spot within the grid has a 10% chance of turning into mud. Only tiles adjacent to the agent can turn into mud. Second, beacons might turn off due to external factors. There is a 5% chance after each action is taken by the agent for a lit beacon to be turned off. When the discrepancy is detected, the agent generates a policy to turn on the beacon (the `relight` action can be taken from anywhere on the grid) and complete the rest of the problem. All actions have a cost of 1, with the exception of `unstuck` which costs 5. Fuel consumption matches the cost of the action.

Figure 6(a) shows the accumulated costs to turn on the 3 beacons. Informed Expectations had the largest action cost to turn on the 3 beacons. This is due to the fact that the plan including Informed Expectations had no ability to avoid the mud patches; so it frequently needs to perform the unstuck action. Immediate, Regression, and G-Regression expectations take into account preconditions of future actions. Since the movement actions have preconditions of the spot being clear (i.e., without mud), a discrepancy is triggered if a necessary tile turns to mud. G-Regression has a slightly higher action cost compared to Policy Regression and Immediate Expectations but as shown in Figure 6(b), Policy Regression and Immediate expectations had extremely high failure rates. This means the agent reached a terminal state without all 3 beacons having been turned on. They don't check for the persistence of effects from previous actions taken by the agent. Goal Regression Expectations triggers a discrepancy if a Beacon is turned back off..

8. Related Work

Goal regression determines the minimal preconditions needed to execute a plan Reiter (1991) and has been used for plan reuse Veloso and Carbonell (1993). Goal regression has also been used to avoid unnecessary replanning Fritz and McIlraith (2007), further extended for dealing with unexpected events in Fritz and McIlraith (2009b), and subsequently for domains with random variables such as the price of the stock market Fritz and McIlraith (2009a). All these works assume solutions to planning problems to be a sequence of actions unlike policies in our work.

For FOND planning, goal regression has been used to bias the policy generation process Ramirez and Sardina (2014). Goal regression plays a central role in the PRP planner Muise, McIlraith, and Beck (2012); it incrementally builds a solution policy by aggregating so-called weak plans: sequences of actions from the start state to a goal state. Goal regression is applied on the weak plans to generate the necessary conditions needed to generate that weak plan. This was further extended to deal with conditional effects computed over the weak plans Muise, McIlraith, and Belle (2014). Whereas in these works regression is performed over action sequences (i.e., the weak plans), in our work we are defining regression for on fully formed policies.

9. Conclusions

We introduce Policy Regression Expectations, which is defined based on the possible trajectories backwards from the terminal states. We also introduce the notion of execution policy enabling continuing execution even when a state visited doesn't match any of the policy's states but match their expectations. We report on a comparative study of policy regression versus immediate and informed expectations adapted for probabilistic domains, and to policy regression without goals expectations. We performed experiments on the Arsonist and the Marsworld domain. Goal Regression and Informed expectations are the only ones guaranteeing the agent to reach a terminal state without failures (i.e., goals are achieved). However, informed expectations do so by having the higher costs than Goal Regression expectations.

For future work, we plan to explore situations where the probability distributions of the ND effects are unknown and statistical learning techniques are used to learn these distributions online. The challenge in that context is that the agent will be operating with (possibly poor) approximations of the probability distributions. This will require the agent to also reason with confidence levels of its own expectations.

Acknowledgements.

This research was supported by ONR under grants N00014-18-1-2009 and N68335-18-C-4027 and NSF grant 1909879.

References

Aha, D. W. 2018. Goal reasoning: foundations emerging applications and prospects. *AI Magazine*.

Cormen, T.; Leirson, C.; Rivest, R.; and Stein, C. 2001. *Introduction to Algorithms*. MIT Press.

Cox, M. T., and Ram, A. 1999. Introspective multistrategy learning: On the construction of learning strategies. *Artificial Intelligence* 112(1-2):1–55.

Cox, M. T.; Dannenhauer, D.; and Kondrakunta, S. 2017. Goal operations for cognitive systems. In *AAAI*, 4385–4391.

Cox, M. T. 2007. Perpetual self-aware cognitive agents. *AI magazine* 28(1):32.

Dannenhauer, D., and Munoz-Avila, H. 2015. Raising expectations in gda agents acting in dynamic environments. In *IJCAI*, 2241–2247.

Dannenhauer, D.; Munoz-Avila, H.; and Cox, M. T. 2016. Informed expectations to guide gda agents in partially observable environments. In *IJCAI*, 2493–2499.

Dannenhauer, D. 2018. *Self Monitoring Goal Driven Autonomy Agents*. Ph.D. Dissertation, Lehigh University.

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence (AMAI)* 18:69–93.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In *ICAPS*, volume 6, 212–221.

Fritz, C., and McIlraith, S. A. 2007. Monitoring plan optimality during execution. In *ICAPS*, 144–151.

Fritz, C., and McIlraith, S. 2009a. Computing robust plans in continuous domains. In *Nineteenth International Conference on Automated Planning and Scheduling*.

Fritz, C., and McIlraith, S. A. 2009b. Generating optimal plans in highly-dynamic domains. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, 177–184. AUAI Press.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gregg-Smith, A., and Mayol-Cuevas, W. W. 2015. The design and evaluation of a cooperative handheld robot. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, 1968–1975. IEEE.

Kuter, U., and Nau, D. S. 2005. Using domain-configurable search control for probabilistic planning. In *National Conference on Artificial Intelligence (AAAI)*, 1169–1174.

Mason, G. R.; Calinescu, R. C.; Kudenko, D.; and Banks, A. 2017. Assured reinforcement learning for safety-critical applications. In *Doctoral Consortium at the 10th International Conference on Agents and Artificial Intelligence*. SciTePress.

Molineaux, M., and Aha, D. W. 2014. Learning unknown event models. In *AAAI*, 395–401.

Molineaux, M.; Klenk, M.; and Aha, D. W. 2010. Goal-Driven Autonomy in a Navy Strategy Simulation. In *AAAI*.

Molineaux, M.; Kuter, U.; and Klenk, M. 2012. Discoverhistory: Understanding the past in planning and execution. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 989–996. International Foundation for Autonomous Agents and Multiagent Systems.

Muise, C. J.; McIlraith, S. A.; and Beck, C. 2012. Improved non-deterministic planning by exploiting state relevance. In *Twenty-Second International Conference on Automated Planning and Scheduling*.

Muise, C.; McIlraith, S. A.; and Belle, V. 2014. Non-deterministic planning with conditional effects. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*.

Muñoz-Avila, H.; Jaidee, U.; Aha, D.; and Carter, E. 2010. Goal-Driven Autonomy with Case-Based Reasoning. In *Case-Based Reasoning. Research and Development*. Springer. 228–241.

Munoz-Avila, H.; Dannenhauer, D.; and Reifsnyder, N. 2019. Is everything going according to plan? - expectations in goal reasoning agents. In *Proceedings of AAAI-19*.

Paisner, M.; Maynord, M.; Cox, M. T.; and Perlis, D. 2013. Goal-driven autonomy in dynamic environments. In *Goal Reasoning: Papers from the ACS Workshop*, 79.

Pollock, J. L. 1998. The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence* 106(2):267–334.

Ramirez, M., and Sardina, S. 2014. Directed fixed-point regression-based planning for non-deterministic domains. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*.

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, (Ed.)*. Academic Press.

Veloso, M. M., and Carbonell, J. 1993. Derivational analogy in PRODIGY: Automating case acquisition, storage and utilization. *Machine Learning* 10(3):249–278.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Munoz-Avila, H. 2007. Adaptation of hierarchical task network plans. In *FLAIRS conference*, 429–434.

Appendix 1. Preliminaries

A state s is a mapping $s : V \to C$, instantiating each variable to a constant. S denotes the collection of all states. A partial function $f : V \nrightarrow C$, is mapping a subset of V, denoted by V_f, such that for each $v \in V_f$, $f(v) \in C$. If $v \in V - V_f$, then $f(v)$ is undefined. A goal is a partial mapping $G : V \nrightarrow C$. Any solution π is correct if for every $v \in V_G, s(v) = G(v)$(s a terminal state of π).

An action is a $a = (name^a, pre^a, Eff^a)$ triple where pre^a is a partial mapping $pre^a : V \nrightarrow C$. Eff^a is a finite set of possible effects and a probability distribution among these effects. Specifically, if $Eff^a = \{(eff^a_{s_1}, P^a_{s_1}) \ldots (eff^a_{s_k}, P^a_{s_k})\}$, then each $eff^a_{s_i}$ is a partial mapping $eff^a_{s_i} : V \nrightarrow C$ and $P^a_{s_i}$ is the probability that the outcome of a is $eff^a_{s_i}$. An operator is a lifted action (see Table 1 for examples). Applying $eff^a_{s_i}$ to a state s, results in a state s_i defined as follows: (1) If $v \in V_{eff^a_{s_i}}$ then $s_i(v) = eff^a_{s_i}(v)$. (2) If $v \notin V_{eff^a_{s_i}}$ then $s_i(v) = s(v)$ (i.e., the variable's value remains unchanged).

A **planning problem** $\mathcal{P}$ is defined as a triple (s_0,$\mathcal{A}$,$\mathcal{G}$), indicating the initial state, the actions and the the goals respectively. A policy $\pi : S \nrightarrow \mathcal{A}$, a partial mapping from the possible states in the world S to actions $\mathcal{A}$. A policy π is a solution to $\mathcal{P}$ with probability $\rho \in (0, 1]$ if when π is executed from s_0 it reaches a state s satisfying $\mathcal{G}$ with probability ρ. Figure 2 is an example of an policy to the problem defined by Table 2 using the operators in Table 1.

Appendix 2. Constructing the Plan Tree T_π

We construct the **Plan Tree** T_π= (V_T, E_T) with root s_0 for $G(\pi) = (V_\pi, E_\pi)$ as shown in Algorithm 1. The initial call is: CONSTRUCTTREE($G(\pi)$, s_0, $\emptyset$) and $E_T = V_T = \emptyset$ holds.

Algorithm 1

```
1: procedure ConstructTree($G(\pi)$,$\alpha$, $Edges$)
2:     for $e = (\alpha, \alpha') \in G(\pi)$ and $e \notin Edges$ do
3:         $E_T = E_T + e$
4:         $V_T = V_T + \alpha + \alpha'$
5:         if $e$ is a branching edge then
6:             $Edges = Edges + e$
7:             $ConstructTree(G(\pi), \alpha', Edges)$
8:         else
9:             $ConstructTree(G(\pi), \alpha', Edges)$
```

The $Edges$ parameter in $ConstructTree$ accumulates all **branching edges**, that is, edges of the form: $(\alpha, v_1) \ldots (\alpha, v_m)$ (i.e., two or more edges starting from the same source α), observed while constructing T_π. $Edges$ is used so that the algorithm traverses any branch, (α, v_k), at most once on each path explored. This prevents any infinite loops from forming while also ensuring that all paths from the starting state to a terminal node are accounted for. *ConstructTree* iterates through all edges in $G(\pi)$ with source α that are not present in $Edges$, adding each edge e to E_T and its source and end vertices, α and α', into V_T (Steps 3 and 4). If e is a branching edge (Step 5), it is added to $Edges$ (Step 6), and T_π is recursively built from e's end, α' (Step 7). If e is not a branching edge, the algorithm is recursively called from α' (Step 9).

Figure 3 showcases a resulting plan tree for $G(\pi)$ in Figure 2. An example of an expanded branching edge is the edge $e = (stack(4, 5), s_{0_1})$. We again expand the subtree rooted at s_{0_1} and point the edge from $stack(4, 5)$ to the new subtree. Edge e is left out of this new subtree. When looking at the right of Figure 3, we can see a pruned tree that ends in the terminal nodes. The edge $e_b = (stack(1, 2), S_2)$ is the last back edge to be expanded on this path, as the back edges $e_1 = (stack(2, 3), S_1)$ and $e_2 = (stack(1, 2), S_2)$ have been pruned on the path following e_b.

Appendix 3: Regression Expectations with Three Composite Operators

We introduce three composite operators that are used to define goal regression precisely. Specifically, they propagate backwards conditions on T_π.

We define $D = A \ominus B$, for $A : V \nrightarrow C$ and $B : V \nrightarrow C$ as a partial function $D : V \nrightarrow C$ defined as follows: (1) If $v \in V_A - V_B$ then $D(v) = A(v)$. (2) For all other variables D is undefined:

$V_D = V_A - V_B$. Informally, $A \ominus B$ takes two partial functions, and creates a new partial function that is defined for all variables from A which are not defined in B, and keeps the values from A.

We define $D = A \oslash k$, for $A : V \nrightarrow 2^{C\times[0,1])}$ and $k \in \mathbb{Z}_+$ as a partial function $D : V \nrightarrow 2^{C\times[0,1]}$ defined as follows: (1) if $v \in V_A$: for every $(c,p) \in A(v)$, then $(c, p * k) \in D(v)$. (2) For all other variables, D is undefined (i.e., $V_D = V_A$) Informally, $A \oslash k$ takes the partial function A and multiplies all probabilities in its probability distribution of constants from variables by the probability k (which is in the range [0,1]).

We define $D = A \otimes B$, for $A : V \nrightarrow 2^{C\times[0,1])}$ and $B : V \nrightarrow 2^{C\times[0,1])}$ as a partial function $D : V \nrightarrow 2^{C\times[0,1]}$ defined as follows: (1) If $v \in V_B - V_A$ then $D(v) = B(v)$. (2) If $v \in V_A - V_B$ then $D(v) = A(v)$. (3) If $v \in V_A \cap V_B$: (a) For every $(c,p) \in A(v)$ where $(c,p') \notin B(v)$, $(c,p) \in D(v)$. (b) For every $(c,p) \in B(v)$ where $(c,p') \notin A(v)$, $(c,p) \in D(v)$. (c) For every $(c,p) \in A(v)$ where $(c,p') \in B(v)$, $(c, p+p') \in D(v)$. (4) For all other variables D is undefined: $V_D = V_A \cup V_B$. Informally, $A \otimes B$ takes two partial functions, A and B, and aggregates the values for all variables from both A and B. When a value for a variable in both A and B share a constant, we add the probabilities together.

Example. These three operators are combined to regress the expectations shown in Figure 4. In the figure, an operator is applied to a state A. The operator has a single precondition, $\{v' : (a, 1.0)\}$ and two ND effects, each with a probability 0.5: $v' = d$ and $v' = e$, reaching states B and C respectively. First we use $\ominus$ to subtract out the preconditions of the action from the children node expectations, i.e., $reg'_B = reg_B \ominus pre^A = \{v : (b, 1.0\}$ and $reg'_C = reg_C \ominus pre^A = \{v : (c, 1.0)\}$ Then we use $\oslash$ to scale the expectations by how many children there are, i.e $reg''_B = reg'_B \oslash 2 = \{v : (b, .5)\}$ and $reg''_C = reg'_C \oslash 2 = \{v : (c, .5)\}$. Lastly, we use $\otimes$ to aggregate the the modified children expectations and the preconditions to calculate the parents expectations, i.e., $reg_A = pre_A \otimes reg''_B \otimes reg''_C = \{v' : (a, 1.0), v : (b, .5), v : (c, .5)\}$.

We define regression expectations, $NX^{pol}_{regress(\pi,s)}$, for a policy, π as $NX^{pol}_{regress(\pi,s)} = reg^{pol}_s$. The variable reg^{pol}_s is a partial mapping $reg^{pol}_s : V \nrightarrow 2^{C\times[0,1]}$ that maps variables to a probability distribution of constants. For example, in Figure 4, $NX^{pol}_{regress(\pi,A)} = \{(a, 1.0), v : (b, .5), v : (c, .5)\}$. We compute reg^{pol}_s over T_π as follows:

(1) If s is a terminal non-goal state, then $reg^{pol}_s = (\varnothing, 1.0)$. The $\varnothing$ expectation indicates the probability the plan will terminate in a non-goal state, thus this expectation states we will terminate in a non-goal state with a 100% probability. (2) If s is a terminal goal state and $\mathcal{G} = \{g_1, \ldots, g_m\}$, then we can define Policy Goal Regression $reg^{pol}_T = \{(g_1, 1.0), \ldots, (g_m, 1.0)\}$.[3] This condition is saying that in the terminal state the agent expects all goals to be achieved with 100% probability. (3) For every action $\pi(s)$, we define $g_{pre^{\pi(s)}} : V \nrightarrow 2^{C\times 1}$ that for every variable v defined in $pre^{\pi(s)}$, i.e., $pre^{\pi(s)}(v) = c$, defines $g_{pre^{\pi(s)}}(v) = \{(c, 1.0)\}$. This indicates that the agent expects that each of the preconditions of the actions to be true with 100% probability. (4) If s is not a terminal state, reg^{pol}_s is defined recursively as follows: For a state s, let $\{s_i \ldots s_k\}$ be the children of $\pi(s)$. We define: $reg^{pol}_s = g_{pre^{\pi(s)}} \otimes ((reg^{pol}_{s_i} \ominus eff^{\pi(s)}_{s_i} \ominus g_{pre^{\pi(s)}}) \oslash P^{\pi(s)}_{s_i}) \otimes \ldots \otimes ((reg^{pol}_{s_k} \ominus eff^{\pi(s)}_{s_k} \ominus$

[3] If the goals are unknown, i.e., $\mathcal{G} = \emptyset$, then $reg^{pol}_T = \emptyset$.

$g_{pre^{\pi(s)}}) \oslash P^{\pi(s)}_{s_k})$ where, for $i \leq j \leq k$: (1) $reg^{pol}_{s_j}$ is the regressed expectation for the child s_j of $\pi(s)$; (2) $eff^{\pi(s)}_{s_j}$ are the ND effects of $\pi(s)$ that result in s_j with a probabilities $P^{\pi(s)}_{s_j}$.

Appendix 4: Properties

Theorem 1. Algorithm 1 terminates and the resulting T_π is unique.

Proof sketch. Termination is guaranteed because every loop in $G(\pi)$ can be attributed to a branching edge. Taking an arbitrary node v, the algorithm will either have to choose an edge $e' = (v, v')$ that will loop back or to continue with a different branch $e'' = (v, v'')$ towards a terminal node, thus making both e' and e'' branching edges. Since e' is a branch, it will be cut off after one iteration thus removing the loop and allowing the algorithm to terminate. To prove uniqueness, the crucial observation is that regardless of the order in which an edge $e_{uv} = (u, v)$ is visited in Line 2 of Algorithm 1, the subtree T_v rooted in v will be the same. The reason for this is that the only modifier for the loop is the set $Edges$, which comes directly from the parent node, and is unaffected by the expansion of the other children. Specifically, if there is a node a with branching edges $e_1 = (a, v_1), e_2 = (a, v_2)$ such that both v_1 and v_2 have a path to a node v with a back edge $e_b = (v, x)$, e_b will be expanded when the algorithm traverses the path from v_1 to x and expanded again when the algorithm traverses the path from v_2 to x. This is regardless of whether v_1 or v_2 is expanded first.

Theorem 2. Let $G(\pi) = (E, V)$ and T_π be its policy tree, then the size of T_π is bounded by $|E||V|(|V| + 1)/2$.

Proof Sketch. Each branching edge can add at most $|V|(|V| + 1)/2$ vertices to T_π. This worst case happens in a fully connected graph, where all edges are branching edges. The first time, an edge $e = (u, v)$ is visited, e is added to $Edges$, and in the recursive call, in the worst case, all vertices are added to T_π (since the graph is fully connected). The second time the algorithm is expanding from u along each path, edge e will not be expanded since it is in $Edges$, which makes v no longer a child of u, thus it will add at most $|V| - 1$ vertices to T_π. This repeats recursively until only 1 edge is added to T_π and there are no more edges that can be expanded. This summation is equal to $|V|(|V| + 1)/2$ and can occur for each edge, resulting in $|E||V|(|V| + 1)/2$ in the worst case.

Theorem 3. T_π subsumes all other trees derived from π.

Proof Sketch. Let T' be a tree derived from π and P' a complete path in T' with expectation $X(T', s)$ for the first occurrence of state s in P', we will prove that there is a complete path P in T_π such that for the first occurrence of state s in P, $X(\pi, s) = X(T_\pi, s) = X(T', s)$ holds. The only complete paths in some tree T' that aren't directly represented in T_π are paths that take a back edge more than once. Consider one such complete path P' in T' that takes a back edge $(\pi(v), v')$ more than once. The reason why the expectations for P' are still represented in T_π is that taking $(\pi(v), v')$ a second time, the procedure results in the same expectation set as when the procedure follows the back edge the first time. This can be seen first in the simple case that the back edge points from a node to itself (i.e., $(\pi(v), v)$). This means that the action $\pi(v)$ taken in state v had no changes made to state v, which is why it branches back to v. Thus, it doesn't change the expectation set. When expanding this example to the back edge $(\pi(v), v')$ leading to a path $v\ \pi(v)\ v'\ \pi(v')\ \ldots\ v$ back to to v, we can observe the same fact. This is because by returning to v, we have assigned the same values to variables in state v, thus in effect what amounts to a no-op action over multiple actions.[4]

[4] It is still necessary to traverse the path $v\ \pi(v)\ v'\ \pi(v')\ \ldots\ v$ the first time, so the procedure can compute expectations for those states visited in the path.