Computing Policy's Regression Expectations in Probabilistic Domains

Noah Reifsnyder, Hector Munoz-Avila
Lehigh University

Abstract

In this paper we investigate the problem of computing regression expectations, needed for execution monitoring, when the course of action being monitored is represented as a policy. Until now, the notion of expectation 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. Policies are used when acting in probabilistic domains, where executing actions might have multiple outcomes with known probability distributions.

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 central theme in research on planning 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: (1) **Detection**: determining if the agent is deviating from the course of action plotted in π; (2) **Correction**: taking corrective action when such a deviation is detected. There are multiple approaches to address the correction problem including 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); goal reasoning, where the agent performs a meta-reasoning process and formulates a new goal (Aha 2018; Cox, Dannenhauer, and Kondrakunta 2017); 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).

To address the detection problem, agents compute the expectation, $X(\pi, a)$, as a function of the next action $a \in \pi$ to be executed and check if this expectation is met in the observed environment, $O(s)$.

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 $O(s)$ (i.e., to define $X(\pi, a)$= *"the preconditions of a"*) 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 $X(\pi, a) = s$). This is how expectations are typically 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 (i.e., $s \neq O(s)$) 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 (Dannenhauer, Munoz-Avila, and Cox 2016; Dannenhauer and Munoz-Avila 2015). The central question that we propose to address in this paper is the following:

When monitoring the execution of a policy π, how to compute the expectations, $X(\pi, a)$, of the next action a execute?

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 it was not preplanned 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. 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.

In this paper, we re-examine the notion of expectations where solutions achieving the goals are policies, which are mappings from states to actions. Policies are used in probabilistic domains, where actions may have multiple possible outcomes. A policy accounts for all foreseen states that the agent might encounter based on the action definitions and the starting state. However, agents might encounter situations it has not planned for. 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 it 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 formalization of goal regression for policies; (2) Results demonstrating the soundness of our formalization as well as low-polynomial running time to compute these expectations; and (3) 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; Dannenhauer, Munoz-Avila, and Cox 2016).

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)* $\leftarrow$ *none*); $stack(?b1, ?b2)$ requires that no blocks are on top of $?b1$ and $?b2$ and has three probabalistic outcomes; either block $?b1$ is placed on top of block $?b2$, block $?b1$ falls to the floor (i.e., with the assignment *floor(?b1)* $\leftarrow$ *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 (Traverso and Pistore 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.

(: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

Consider an example, when the initial state starts with 5 blocks (see Table 2): 1, 2, 3, 4, 5 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.

(: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 :Costs $c(S, a) = 1$ for all a and all S :Rewards $R(S_4) = 1$ $R(S) = 0$; if $S \neq S_4$)

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.

Figure 1 shows an example of a **policy** solving this problem. We will define policies carefully in the next section. But briefly, 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.

Preliminaries

In the state-variable representation, a variable $v \in V$ can take a value from C, a set of constants. For instance, in Table

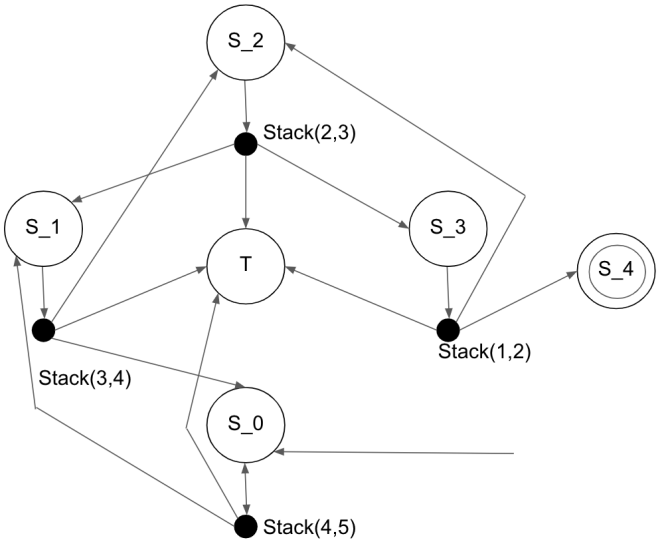

Figure 1: A policy for given state from Table 2 and operator definitions in Table 1. s_0 is the initial state, and s_4 is the terminal state that achieves the goals. T represents a terminal state at which the policy fails to achieve the goals. Actions are represented by the black dots.

2, there are 20 variables: 5 for onfire, 5 for floor, 5 for above, and 5 for below; they each are mapped to a constant. A state s is a mapping $s : V \to C$, instantiating each variable to a constant. S denotes the collection of all states. Exemplified by Table 2, we can see that *onfire(1)* = *False* in the initial state, where the variable v is *onfire(1)* and the function is the state, i.e, *s(onfire(1))= False*. A reward is a mapping $r : S \to \mathbb{R}$ that denotes how favorable a state is. A cost is a mapping $c : A \to \mathbb{R}$ that denotes how much a action "costs" an agent to take.

We use partial functions $f : V \nrightarrow C$, mapping a subset of V, denoted by V_f, such that for each $v \in V_f$, $f(v) \in C$. The function is undefined for all other variables: if $v \in V - V_f$, then $f(v)$ is undefined. These partial functions are used in the definitions of action' preconditions and effects because they only refer to a subset of the variables in the state. For instance, the unstack(1,2) action, the preconditions only mention variables related to the two blocks 1 and 2; the conditions for all other functions are unspecified.

In a probabilistic domain, operators have multiple possible effects: An operator $a = (name^a, pre^a, Eff^a)$ 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 $Eff^a = \{(eff^a_{s1}, P^a_{s1}) \dots (eff^a_{sk}, P^a_{sk})\}$, then each eff^a_{si} is a partial mapping $eff^a_{si} : V \nrightarrow C$ and P^a_{si} is the probability that the outcome of a is eff^a_{si}. Table 1 shows an example of two operators.

An action is a grounded instance of an operator. An action a is applicable in a state s if for every variable $v \in V_{pre^a}$, $pre^a(v) = s(v)$. *app(s)* denotes the set of all actions applicable in state s. Applying a to s will choose one of the effects in Eff^a, based on their probability distribution, and result in a state s_i. In this case, we say that a ND choice is made and that $s_i \in ND(a, s)$. The states and actions define a transition function, $\Gamma : S \times A \to 2^S$, where $\Gamma(s, a)$ is defined if $a \in app(s)$, in which case $\Gamma(s, a) = ND(a, s)$. Applying eff^a_{si} to a state s, results in a state s_i defined as follows:

- if $v \in V_{eff^a_{si}}$ then $s_i(v) = eff^a_{si}(v)$
- If $v \notin V_{eff^a_{si}}$ then $s_i(v) = s(v)$ (i.e., the variable's value remains unchanged).

Example: With the initial state s_0 as in Table 2, then applying $eff^{stack(4,5)}_{s1}$ to s_0 results in $s_1 =$ {above: {1:None, 2:None, 3:None, 4:None, **5:4**}, below: {1:None, 2:None, 3:None, **4:5**, 5:None}, onfire: {1:False, 2:False, 3:False, 4:False, 5:False}, floor: {1:False, 2:False, 3:False, 4:False, 5:False}}

A policy $\pi : S \nrightarrow \mathcal{A}$, a partial mapping from the possible states in the world S to actions $\mathcal{A}$, indicating for any given state $s \in S_\pi$, what action $\pi(s)$ to take. States in $S - S_\pi$ are not reachable from s_0. An **execution trace** is any sequence $s_0\ \pi(s_0)\ \dots\ \pi(s_n)\ s_{n+1}$, where $s_i \in ND(\pi(s_{i-1}), s_{i-1})$. A reachable state s is **terminal** if $\pi(s)$ is not defined.

A **planning problem** $\mathcal{P}$is defined as a triple (s_0,$\mathcal{G}$,$\mathcal{M}$), indicating the initial state, the goals and the MDP respectively. The goals $\mathcal{G}$ are a partial mapping $\mathcal{G}$: $V \nrightarrow C$. $\mathcal{G}$ are **satisfied** in a state s if for every variable $v \in V_\mathcal{G}$, $\mathcal{G}(v) = s(v)$. Any state where the $\mathcal{G}$ is satisfied is a terminal state.

An MDP $\mathcal{M}$=*(S,A,P,*Γ*,C,R)*, where S, $\mathcal{A}$, Γ are the states, actions and transition function as defined before; P are the collection of probability transitions P^a_s, also defined before; $c : S \times A \to [0, \infty)$ is a cost function and $R : S \to [0, \infty)$ is a reward function. For every state $s \in S$, the value of an state $V(s)$ is defined as follows:

- $V(s) = R(s)$, when s is terminal.
- $V(s) = max_{a \in app(s)} Q(s, a)$

  where ($\gamma \in [0, 1]$ is the discount factor):

$$Q(s, a) = R(s) - C(s, a) + \gamma \sum_{s' \in ND(a,s)} P^a_{s'} V(s')$$

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 an state s satisfying $\mathcal{G}$ with probability ρ. A policy π is an optimal solution to $\mathcal{P}$ with probability ρ if (1) π is a solution to $\mathcal{P}$ with a probability ρ, (2) for any other solution π' to $\mathcal{P}$ with a probability ρ, $V_\pi(s_0) \geq V_{\pi'}(s_0)$, and (3) there exists no solution to $\mathcal{P}$ with a probability in range $(\rho, 1]$.

Figure 1 is an example of an optimal policy to the problem defined by Table 2 using the operators in Table 1. We define $G(\pi) = (V_\pi, E_\pi)$ as the graph form of the policy, as visualized by Figure 1

Regression Policy Expectations

To define regression on a policy π, we perform two steps: (1) Compute the plan tree T_π ; and (2) Compute regression on T_π.

Plan Tree T_π

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

Algorithm 1

```
1: procedure CONSTRUCTTREE(G(π),α, Edges)
2:     for e = (α, α') ∈ G(π) and e ∉ Edges do
3:         E_T = E_T + e
4:         V_T = V_T + α + α'
5:         if e is a branching edge then
6:             Edges = Edges + e
7:             ConstructTree(G(π), α', Edges)
8:         else
9:             ConstructTree(G(π), α', Edges)
```

The $Edges$ parameter in $ConstructTree$ accumulates all **branching edges**, that is, edges of the form: $(\alpha, v_1) \dots (\alpha, v_m)$ (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 2 showcases a resulting plan tree for $G(\pi)$ in Figure 1. An example of an expanded branching edge is the edge $e = (stack(4,5), s_0)$. We again expand the subtree rooted at s_0 and point the edge from $stack(4,5)$ to the new subtree. Note that e is left out of the new subtree.

Three Composite Operators

We introduce three composite operators that are used to propagate backwards conditions on T_π.

The first operator, $\ominus$, takes the intersection of two partial mappings, choosing one set of mappings over the other if the mapping of a key differs between the two sets of mappings. 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. For example, if $A = \{onfire : \{1 : True, 2 : False\}\}$ and $B = \{above : \{5 : 4\}, onfire : \{2 : True, 3 : True\}\}$, then $A \ominus B = \{onfire : \{1 : True\}\}$.

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]). For example, if $A = \{above : \{5 : (4,1), 4 : (3,.2)\}, onfire : \{1 : (True, .5)\}\}$, then $A \oslash .5 = \{above : \{5 : (4,.5), 4 : (3,.1)\}, onfire : \{1 : (True, .25)\}\}$. $\oslash$ is used to project the probabilities among the probability of the actions' ND effects.

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. When using $\otimes$, our formulas guarantee that a probability distribution will remain within [0,1] (by being used in conjunction with $\oslash$). For example, if $A = \{above : \{5 : (4,1), 4 : (3,.2)\}, onfire : \{1 : (True, .5)\}\}$ and $B = \{above : \{5 : (4,.5), 4 : (3,.5)\}, onfire : \{1 : (True, .1)\}\}$, then $A \otimes B = \{above : \{5 : (4, 1.5), 4 : (3,.7)\}, onfire : \{1 : (True, .6)\}\}$. While the resulting $above(4)$ value looks to exceed the range of probabilities, this would be fixed through the use of an $\oslash$, i.e $(A \otimes B) \oslash .5)$

Policy Regression Expectations

We define regression expectations formally as $NX^{pol}_{regress}(\pi, s, \emptyset) = reg^{pol}_s$, where 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. We compute reg^{pol}_s over T_π as follows:

- If s is a terminal non-goal state, then $reg^{pol}_s = (\oslash, 1.0)$. The $\oslash$ 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

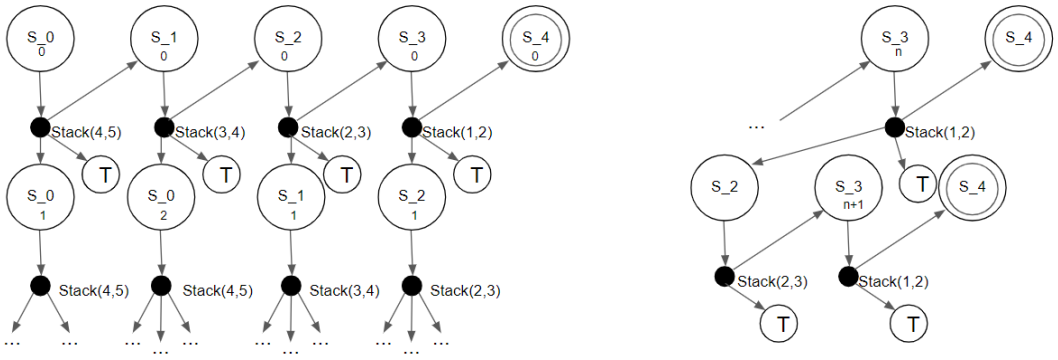

Figure 2: The expanded plan tree T_π used for calculating policy goal regression expectations. The numbers next to vertexes are the order that vertex is visited in our search.

- If s is a terminal goal state, then $reg_s^{pol} = \emptyset$. This indicates we don't know the goals that are satisfied.[1]
- 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 $g_{pre^{\pi(s)}}(v) = \{(c, 1)\}$. This is an auxiliary function indicating that the agent expects that each of the preconditions of the actions to be true with 100% probability.
- If s is not a terminal state, reg_s^{pol} is defined recursively as follows: For a state s, let $\{s_i ... s_k\}$ be the children of $\pi(s)$. We define $reg_s^{pol} = g_{pre^{\pi(s)}} \otimes ((reg_{s_i}^{pol} \ominus eff_{s_i}^{\pi(s)} \ominus g_{pre^{\pi(s)}}) \oslash P_{s_i}^{\pi(s)}) \otimes ... \otimes ((reg_{s_k}^{pol} \ominus eff_{s_k}^{\pi(s)} \ominus g_{pre^{\pi(s)}}) \oslash P_{s_k}^{\pi(s)})$, where, for $i \le j \le k$: (1) $reg_{s_j}^{pol}$ is the regressed expectation for the child s_j of $\pi(s)$; (2) $eff_{s_j}^{\pi(s)}$ are the ND effects of $\pi(s)$ that result in s_j with a probabilities $P_{s_j}^{\pi(s)}$. Informally, the agent computes the expectations for an state by regressing the (regressed) expectations of its children, weighted by the probability of reaching each child.

If s occurs more than once in T_π, then we define $NX_{regress}^{pol}(\pi, s, \emptyset)$ 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 2, s_0 occurs multiple times and the one selected is the root.

We show how to calculate $reg_{s_3}^{pol}$. In T_π there are numerous occurrences of s_3 (of which Figure 2 shows three). For sake of space we will focus on the occurrence labelled n. $\pi(s_3) =$ *stack(1,2)* has three children, T, s_2 and s_4. s_4 is a terminal goal state and always has the same expectations, $reg_{s_4}^{pol} = \emptyset$. s_2 has the expectations (calculation not shown), $reg_{s_2}^{pol} = \{above : \{1 : (None, .9), 2 : (None, 1), 3 : (None, 1)\}, onfire : \{1 : (False, .9), 2 : (False, 1)\}, \varnothing : .038\}$. The expectations for T are $\{\varnothing : 1\}$ $g_{pre^{stack(1,2)}} = \{above : \{1 : (None, 1), 2 : (None, 1)\}, onfire : \{1 : (False, 1)\}\}$.

Thus,
$reg_3^{pol} = g_{pre^{stack(1,2)}}$
$\otimes((reg_{s_2}^{pol} \ominus eff_{s_2}^{stack(1,2)} \ominus g_{pre^{stack(1,2)}}) \oslash P_{s_2}^{stack(1,2)})$
$\otimes((reg_{s_4}^{pol} \ominus eff_{s_4}^{stack(1,2)} \ominus g_{pre^{stack(1,2)}}) \oslash P_{s_4}^{stack(1,2)})$
$\otimes((reg_{T}^{pol} \ominus eff_{T}^{stack(1,2)} \ominus g_{pre^{stack(1,2)}}) \oslash P_{T}^{stack(1,2)})$
and $\emptyset \oslash P_{s_0}^{stack(1,2)} = \emptyset$.
Therefore:

- With $P_{s_2}^{stack(1,2)} = .08$ probability, $stack(1, 2)$ knocks block 2 off the tower. Thus, $(reg_{s_2}^{pol} \ominus eff_{s_2}^{stack(1,2)} \ominus g_{pre^{stack(1,2)}}) = \{onfire : \{2 : (False, 1)\}, \varnothing : .038\}$, and $\{onfire : \{2 : (False, 1)\}, \varnothing : .038\} \oslash .08 = \{onfire : \{2 : (False, .08)\}, \varnothing : .00304\}$.
- With $P_{s_4}^{stack(1,2)} = .9$ probability, *stack(1, 2)* stacks block 1 on top of block 2. $(reg_{s_4}^{pol} \ominus eff_{s_4}^{stack(1,2)} \ominus g_{pre^{stack(1,2)}} = \emptyset$ because $reg_{s_4}^{pol} = \emptyset$. Again $\emptyset \oslash P_{s_4}^{stack(1,2)} = \emptyset$.
- with $P_T^{stack(1,2)} = .02$ probability, $stack(1, 2)$ knocks block 1 to the floor. $(reg_T^{pol} \ominus eff_T^{stack(1,2)} \ominus g_{pre^{stack(1,2)}} = \{\varnothing : 1.0\}$. Then $\{\varnothing : 1.0\} \oslash .02 = \{\varnothing : .02\}$

Now we have $reg_{s_3}^{pol} = g_{pre^{stack(1,2)}} \otimes \{onfire : \{2 : (False, .08)\}, \varnothing : .00304\} \otimes \emptyset \otimes \{\varnothing : .02\}$. Thus $reg_{s_3}^{pol} = \{above : \{1 : (None, 1), 2 : (None, 1)\}, onfire : \{1 : (False, 1), 2 : (False, .08)\}, \varnothing : .02304\}$. So when in state s_3 we would like nothing to be on top of either block 1 or 2, both blocks 1 and 2 not to be on fire (block 2 not being on fire only matters 8% of the time), and there is a 2.304% chance we will end in a non goal terminal state.

Properties

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

[1] If $\mathcal{G} = \{g_1, \ldots, g_m\}$ is known, then we can define Policy Goal Regression $greg_T^{pol} = \{(g_1, 1.0), \ldots, (g_m, 1.0)\}$. This condition is saying that in the terminal state the agent expects all goals to be achieved with 100% probability. When s is not a terminal state, then $greg_s^{pol} = reg_s^{pol}$

Proof sketch. Termination is guaranteed because every loop in $G(\pi)$ can be attributed to a branching edge. At some node v, we will either have to choose the loop or to continue towards a terminal node, thus making the loop a branching edge. Since it 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 $e = (u, v)$ is visited in Line 2 of Algorithm 1, the subtree 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.

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 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 $G(\pi)$.

Policy Execution Monitoring

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 $f_X = NX^{pol}_{reg}(\pi, s_i, \emptyset)$. By definition, $f_X(v)$ is a probability distribution for all values that v may take (where $f_X(\varnothing)$ always returns $False$). We add the probabilities of values that are not equal to value of v in the observed state, $s(v)$: $P = \Sigma_{s(v)\neq c',(c',p)\in f_X(v)}(p)$. Thus, P indicates the percent of children vertices that are no longer reachable if $s(v) = c$. A domain-specific parameter, δ, is needed to trigger a discrepancy. We say that a discrepancy occurs if $P > 0.5$. In our experiments, we set $\delta = 0.5$ as a threshold as it denotes that more than half the remaining tree is no longer reachable and therefore the execution of the policy is more likely to fail.

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 re known (in the figures we call it G-Regression). 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.

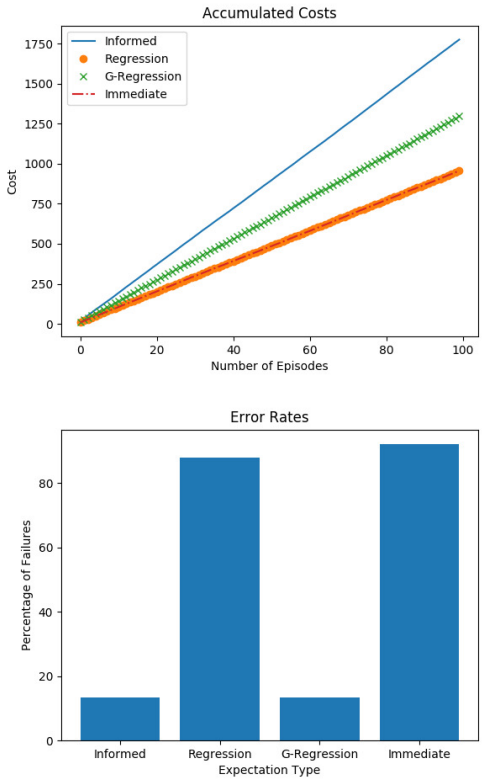

Figure 3: (a) Accumulated costs - Arsonist Domain; (b) Failure rates - Arsonist Domain

For planning, we implemented a probabilistic domain-configurable planner as described in (Kuter and Nau 2005). The policies generated were optimal for the domain. Aside from the different expectations, the agent was the same: it has a simple module that given a discrepancy, it always generates the goal based on a mapping from discrepancies to goals $d \rightarrow g$. It will always generate the same goal g for the same discrepancy d. 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. So if block 2 was on fire from time t=2 to t=4, and block 2 is in the final tower, it contributes a cost of 2 to the final cost) plus the total number of actions taken. 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 has the same probabilistic outcomes as in Table 1. 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. It is possible even with an optimal policy for the agent to finish executing with some blocks on fire, because the agents knowledge of the state is limited by the form of expectation that the agent is using (i.e. the domain is partially observable with respect to the set of expectations the agent is using). The limitation is in place because if we identified states by comparing all state variables, we would be using state expectations. As reported in (Dannenhauer, Munoz-Avila, and Cox 2016). State expectations force a GDA process even in situations when it is unnecessary as they flag any observed inconsistency between the expected and the observed states. The other forms of expectations have been developed to reduce unnecessary activation of a GDA process; this occurs when variables change in the state that do not affect the current course of action.

Figure 3(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 /bf stack actions necessary and then terminate, regardless of if the tower holds 10 blocks or not. This was also not enough time for many blocks to catch on fire therefore they could not accumulate a large cost. Policy Regression and Immediate Expectations had near 100% failure rates (as seen in Figure 3(b)), due to the same lack of knowledge in the expectations. (i.e. at some point the tower is knocked over and was never rebuilt or a final block was on fire). G-Regression had a cost well below informed Expectations, and a failure rate of 13%. 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 13% 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).

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.

In this work we are concerned with the problem of computing expectations for a given policy; not with its generation. The problem of finding the existence of a solution with a probability ρ to a planning problem were actions have probabilistic effects is EXPTIME-hard (Littman 1997). We are assuming episodic tasks; as opposed to continuing tasks when the agent never terminates its execution (Sutton and Barto 2018). We are also assuming that the dynamics of the environment are given in the form of the probability transitions P_s^a. If these are not given, we will need to make assumptions akin to FOND planning (Ramirez and Sardina 2014). For instance assuming an equiprobable distribution in the actions' effects: $Eff^a = \{(eff_{s_1}^a, 1/k) \dots (eff_{s_k}^a, 1/k)\}$.

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.

Conclusions

We introduce Policy Regression Expectations, which is defined based on the possible trajectories backwards from the terminal states. We report on a comparative study of policy regression versus immediate and informed expectations and adapted for probabilistic domains. 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.

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.

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.

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.

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.

Littman, M. L. 1997. Probabilistic propositional planning: Representations and complexity. In *National Conference on Artificial Intelligence (AAAI)*, 748–761. Providence, Rhode Island: AAAI Press / MIT Press.

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.; 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. Nondeterministic 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.

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.

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.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Traverso, P., and Pistore, M. 2004. Automated composition of semantic web services into executable processes. In *ISWC-2004*.

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.