

Frog

Due: 11:59 PM 4/16/08

Introduction

In class, we have introduced you to Markov models, a powerful way to formulate problems involving uncertainty over time. We have also taught you about reinforcement learning, a way to learn the optimal policy for how one should act in a dynamic and stochastic world.

If you think about it, life is a little like a Markov process. Each moment is a chance to start over, independent of the path you took to get where you are. And while you have some choice of the actions you take, the future is uncertain. Life is like sitting on the saddle of a giant, perhaps inebriated frog, trying your hardest to steer it in the direction you want despite the frog having a mind of its own. Success is never guaranteed, but we can try our hardest to find the optimal *life-policy* that is *most likely* to take us to our goal while avoiding life's pitfalls.

Completely coincidentally, that's sort of what this project is about.

Our daring (and darling) TA Joe Austerweil is taking a magic Markov ride through the wumpus world on the back of a giant frog. And he needs your help! Joe needs to collect the mushrooms he needs for his special *Fricasée de Champignons des Bois* recipe as quickly as possible while navigating a hazardously pit-filled world. But that's much harder than it sounds. He discovered his ranine companion this morning sipping fly juice that had been left out in the sun so long that it had *fermented* (unbeknowst, of course to the frog.) What would have been an easily determinist problem has suddenly turned precariously stochastic!

Thankfully, he has you to guide him by modelling his situation as a Markov Decision Process (MDP) and determining his optimal policy using reinforcement learning. Huzzah!

Markov Decision Processes

Let's recap the formal theory behind Markov Decision Processes. A Stochastic Decision Process is defined by

1. a set of states S
2. a set of actions A
3. a reward function $r : S \times A \rightarrow \mathbb{R}$
4. state transition probabilities $P(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots)$

At each state s_t

1. The agent chooses an action a_t
2. The process yields a reward $r(s_t, a_t)$
3. The process transitions to a new state s'_{t+1} with probability $P(s'_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots)$

In a Markov Decision Process, P is assumed to have the markov property

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots)$$

A policy π for an MDP is a map $\pi : S \rightarrow A$.

It tells the agent to take action $\pi(s)$ when at state s .

We often derive policies from a reward estimator $Q : S \times A \rightarrow \mathbb{R}$, which will tell us the estimated reward of taking a particular action at a particular state. To derive the best policy given such an estimator, we let $\pi(s)$ be equal to the action that offers the best expected reward, according to the estimator.

SARSA

In this project, you will work within an MDP policy learning paradigm called *reinforcement learning*. Over alternative policy learning methods, such as policy iteration, it has the benefits of not requiring as much computation over the entire state space and working even when the probabilities in the Markov process are unknown. The way these algorithms work is by stochastically exploring the MDP and updating the estimated values of actions and states by inferring them from the *observed* rewards.

The particular algorithm we want you to implement is called SARSA, and it has the interesting feature of learning *as* the policy is being acted upon. For this reason, it is called an *on-policy* learning algorithm.

SARSA gets its name from the quintuple, $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, which drives the algorithm. The idea is that at each stage of the algorithm t , SARSA simulates action a_t on s_t , gets the reward r_{t+1} , and transitions to s_{t+1} . Then it uses the currently learned policy to choose the next action, a_{t+1} . Finally, we update our reward estimate Q as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

where γ is the discount factor and $0 < \alpha < 1$ decays with time to guarantee convergence.

When choosing the next action according to the currently learned policy π , SARSA is ϵ -greedy. That means that when for a given state s the policy says to perform action $\pi(s)$, SARSA picks action $\pi(s)$ with probability $1 - \epsilon$ but otherwise chooses an action at random. In some implementations, ϵ decays over time.

The algorithm looks like this:

1. Initialize Q, π randomly.

2. Repeat:

- (a) Initialize s to the start state.
- (b) Initialize a to $\pi(s)$ with probability $1 - \epsilon$, otherwise randomly.
- (c) While s is non-terminal:
 - i. Take action a , observing reward r and new state s' .
 - ii. With probability $1 - \epsilon$, let $a' = \pi(s')$, otherwise choose a' randomly.
 - iii. $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
 - iv. $\pi(s) = \arg \max_a Q(s, a)$
 - v. $s \leftarrow s', a \rightarrow a'$
 - vi. Decay α, ϵ according to schedule.

What to Do

Your task is to implement a class `SarsaPolicyRunner` that runs the SARSA algorithm. The javadocs for the interface it implements, `PolicyRunner`, should tell you what to do more precisely.

For your final handin, please calculate your SARSA parameters ϵ , γ , and α like so:

$$\begin{aligned}\epsilon &= \frac{1}{\log 1 + t} \\ \gamma &= 0.8 \\ \alpha &= \frac{1}{\log 1 + t}\end{aligned}$$

You are highly encouraged, however, to play with values for amusement and edification.

What to Handin

Please hand in the following:

1. All of your code (including our support code).
2. A README, including:
 - (a) Things that may help us read your code: design quirks, conventions, etc.
 - (b) Standard bug reporting.
 - (c) Any comments that might help us improve this project for future.

Hand in by typing `cs141handin frog` in the shell from the directory containing your work.

Support Code

Copy the support code from `/course/cs141/src/frog/` to your course directory.

For this assignment we have given you

1. A package `mdp` for creating and running Markov Decision Processes
2. A package `frog` that generates MDPs from provided frog board (`boards/*.fb`) files and displays them in a slick GUI.
3. A package `util` containing some tools that were used in the implementation of the support. You may find it useful as well. Consider giving `Collections` and `MultinomialDistribution` a look. `MultinomialDistribution` will have more features than you need. Think of it as a `HashMap<E,Double>` with an `argmax` method.
4. A package `sarsa` containing a stencil for the class you have to write, `SarsaPolicyRunner`.
5. A package `main` containing a class `Main` with the project's main method.
6. An ant build file. You can use it to run the demo, compile your code, run your code, and generate javadocs. Type `ant help` in the directory with your project files to learn how to use it.

There is a lot of code in this project. You only need to write `SarsaPolicyRunner`. Be myopic.

On frog

The `SarsaPolicyIterator` class you'll be implementing is general enough to work with any MDP. But for this project, we need to use SARSA learning to help Joe in the wumpus world! These notes below should help you unravel the mystery.

States

States are squares on the board.

These states have types `Ground`, `Pit`, `Wall`, and `Mushroom`.

The rule for generating transition probabilities keeps you off the Walls.

There is also a terminal state.

The `Mdp.toString()` method uses "`G(3,4)`" to represent the square at row 3, column 4 when it is a `Ground` square.

It uses "`End`" to represent the terminal state.

This `toString` method may be useful when debugging.

Actions

Actions are screen-directions: Up, Down, Left, and Right.

Transitions

A step in the requested direction is taken with probability $1 - \text{misstepProbability}$ where `misstepProbability` is given on the command line or as a buildfile default.

The probability of each misstep is the same.

Moving into a Wall is considered a step, but doesn't move you.

All transitions from Pits and Mushrooms lead to the terminal state.

Rewards

The reward function we use depends only on state type (it is constant with respect to action).

The reward for each state type is given on the command line or as a buildfile default.

If you want to use other values, it is important to make them reasonable.

To make infinite loops undesirable, the Ground should almost certainly have a negative reward.

The GUI

The purple arrows on the ground display your policy.