Assignment #6

Due: 23:55pm Friday 10 May 2019

Upload at: `https://dropbox.cs.princeton.edu/COS324_S2019/HW6`

Problem 1 (1pt)
Use the instructions above to put your name on the assignment when you compile this document.

Problem 2: Lawn Darts [49 pts]

Lawn darts was a classic fun family game, that also turned out to be somewhat deadly. The game was played by throwing pointy weighted darts into the air and trying to have them land in a target area. It was kind of like a weaponized version of horseshoes. Like the *Gilbert U-238 Atomic Energy Lab*, it is a toy that has been banned by the Consumer Product Safety Commission.

Fortunately, we're going to play it here in simulation. This will be a single-player game. The idea is to throw darts at the grid in Figure 2 until you get **exactly** 101 points. You'll start with zero points, and in each turn you aim a dart at one of the sixteen inner numbered points in the grid. You're not a great aim, so you have a good chance to hit one of the other squares. Whatever square you hit, you add that number of points to your score. If you hit one of the blank squares in the ring, you add zero points. You can't aim for the blank ring. If you get a score of exactly 101, the game ends and you get one unit of reward. If you go over 101, the game ends and you get a negative one unit of reward.

WARNING:
Pointed, metal lawn darts may cause serious or fatal injury.
Parents are urged to discard or
destroy them immediately.

Figure 1: Lawn darts turned out to be a bad idea, except for in simulation.

	7	12	1	14	
	2	13	8	11	
	16	3	10	5	
	9	6	15	4	

Figure 2: You aim for one of the 16 middle squares. You get zero points if you accidentally hit in the ring outside.

Your objective is to learn a policy to play the game. In this case, you happen to know exactly what your aiming distribution is: you have a 60% chance of hitting the square you aim for, and 10% chance of hitting the square to the north, east, south or west. So, if you aim for the 15, you get 15 points with probability 0.6, 4 points with probability 0.1, 6 points with probability 0.1, 10 points with probability 0.1, and 0 points with probability 0.1. This is a Markov decision process (MDP) in which you know the current state $s \in \{0, 117\}$ (your score), you know the reward function

$$R(s) = \begin{cases} 0 & \text{if } s < 101 \\ 1 & \text{if } s = 101 \\ -1 & \text{if } s > 101 \end{cases}, \tag{1}$$

and you know the transition probabilities. A policy for this game will be a function from the states $s = 0, 1, \ldots, 100$ to 16 possible actions. Your objective is to find such a policy using value iteration or policy iteration. Explain how you set up the problem and why you chose the approach you did. Plot a representation of the evolution of the value function over time, that is, make a figure where the x-axis is the current state (score) and the y-axis is the value of that

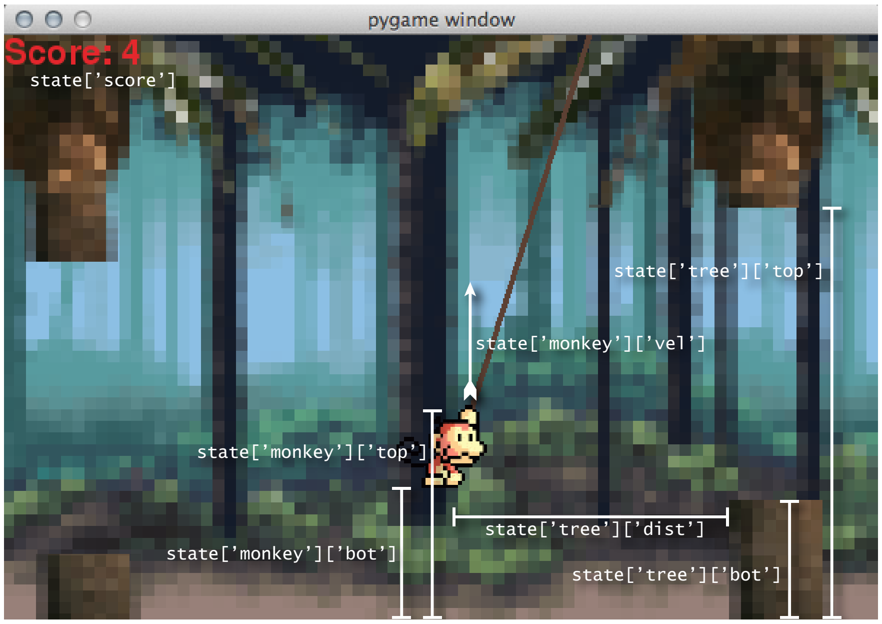

Figure 3: Various pieces of the state dictionary.

state. (You don't need to plot these curves for each iteration of value iteration, just for the final solution.) Do you notice any interesting structure in this plot? Which non-terminal state has the highest value? Why do you think this is? Turn in the code you used to perform this simulation.

Problem 3: Swingy Monkey [50 pts]

In 2014, the mobile game *Flappy Bird* took the world by storm. After its discontinuation, iPhones with the game installed sold for thousands of dollars on eBay. In this problem, you'll be developing a reinforcement learning agent to play a similar game, *Swingy Monkey*. In this game, you control a monkey that is trying to swing on vines and avoid tree trunks. You can either make him jump to a new vine, or have him swing down on the vine he's currently holding. You get points for successfully passing tree trunks without hitting them, falling off the bottom of the screen, or jumping off the top. There are some sources of randomness: the monkey's jumps are sometimes higher than others, the gaps in the trees vary vertically, and the distances between the trees are different. You can play the game directly by pushing a key on the keyboard to make the monkey jump, but this problem is about building a learning agent to play.

You will implement two Python functions, `action_callback` and `reward_callback`. The reward callback will tell you what your reward was in the immediately previous time step:

- Reward of $+1$ for passing a tree trunk.
- Reward of -5 for hitting a tree trunk.
- Reward of -10 for falling off the bottom of the screen.
- Reward of -10 for jumping off the top of the screen.
- Reward of 0 otherwise.

The action callback will take in a dictionary that describes the current state of the game and you will use it to return an action in the next time step. This will be a binary action, where 0 means to swing downward and 1 means to jump up. The dictionary you get for the state looks like this:

```
{ 'score': <current score>,
  'tree': { 'dist': <pixels to next tree trunk>,
            'top':  <height of top of tree trunk gap>,
            'bot':  <height of bottom of tree trunk gap> },
'monkey': { 'vel': <current monkey y-axis speed>,
            'top': <height of top of monkey>,
            'bot': <height of bottom of monkey> }}
```

All of the units here (except score) will be in screen pixels. Figure 3 shows these graphically. There are multiple challenges here. First, the state space is very large – effectively continuous. You'll need to figure out how to handle this. One strategy might be to use some kind of function approximation, such as a neural network, to represent the value function or the Q-function. Another strategy – one that worked well for me – is to discretize the position space into bins. Second, you don't know the dynamics, so you'll need to use a reinforcement learning approach, rather than a standard MDP solving approach. I got a pretty good monkey policy with Q-Learning.

Your task is to use reinforcement learning to find a policy for the monkey that can navigate the trees. The implementation of the game is in file `SwingyMonkey.py`, along with a few files in the `res/` directory. A file called `stub.py` is provided to give you an idea of how you might go about setting up a learner that interacts with the game. I also posted a YouTube video of my Q-Learner at here. You can see that it figures out a reasonable policy in a few dozen iterations. (Don't take the specific performance or number of iterations too seriously as a threshold for a good solution. The goal is to get a competent monkey policy; your algorithm might find one faster or slower than mine. The 100 iterations in the starter code is not a constraint.) You should explain how you decided to solve the problem, what decisions you made, and what issues you encountered along the way. Provide evidence where necessary to explain your decisions. Effort and explanation will be large fractions of the grading rubric of this problem. You should not use off-the-shelf RL tools like PyBrain to solve this. Turn in your code.

Changelog

- 29 April 2019 – Initial version.