# Homework 6: Inference in Graphical Models, MDPs

## Introduction

In this assignment, you will practice inference in graphical models as well as MDPs/RL. For readings, we recommend Sutton and Barto 2018, Reinforcement Learning: An Introduction, CS181 2017 Lecture Notes, and Section 10 and 11 Notes.

Please type your solutions after the corresponding problems using this LaTeX template, and start each problem on a new page.

Please submit the **writeup PDF to the Gradescope assignment 'HW6'**. Remember to assign pages for each question.

Please submit your **LaTeX file and code files to the Gradescope assignment 'HW6 - Supplemental'**.

You can use a **maximum of 2 late days** on this assignment. Late days will be counted based on the latest of your submissions.

**Problem 1** (Explaining Away + Variable Elimination 15 pts)

In this problem, you will carefully work out a basic example with the "explaining away" effect. There are many derivations of this problem available in textbooks. We emphasize that while you may refer to textbooks and other online resources for understanding how to do the computation, you should do the computation below from scratch, by hand.

We have three binary variables: rain $R$, wet grass $G$, and sprinkler $S$. We assume the following factorization of the joint distribution:

$$\Pr(R, S, G) = \Pr(R)\Pr(S)\Pr(G \mid R, S).$$

The conditional probability tables look like the following:

$$\begin{aligned}
\Pr(R = 1) &= 0.25 \\
\Pr(S = 1) &= 0.5 \\
\Pr(G = 1 | R = 0, S = 0) &= 0 \\
\Pr(G = 1 | R = 1, S = 0) &= .75 \\
\Pr(G = 1 | R = 0, S = 1) &= .75 \\
\Pr(G = 1 | R = 1, S = 1) &= 1
\end{aligned}$$

1. Draw the graphical model corresponding to the factorization. Are $R$ and $S$ independent? [Feel free to use facts you have learned about studying independence in graphical models.]
2. You notice it is raining and check on the sprinkler without checking the grass. What is the probability that it is on?
3. You notice that the grass is wet and go to check on the sprinkler (without checking if it is raining). What is the probability that it is on?
4. You notice that it is raining and the grass is wet. You go check on the sprinkler. What is the probability that it is on?
5. What is the "explaining away" effect that is shown above?

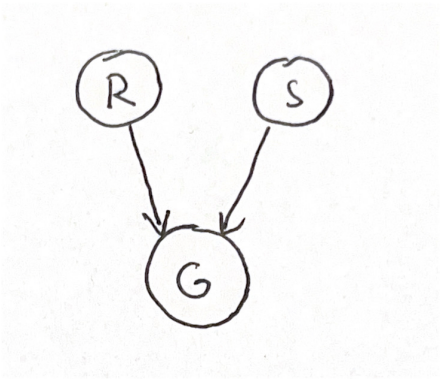
Consider if we introduce a new binary variable, cloudy $C$, to the the original three binary variables such that the factorization of the joint distribution is now:

$$\Pr(C, R, S, G) = \Pr(C)\Pr(R|C)\Pr(S|C)\Pr(G \mid R, S).$$

6. For the marginal distribution $\Pr(R)$, write down the variable elimination expression with the elimination ordering $S, G, C$ (where $S$ is eliminated first, then $G$, then $C$).
7. For the marginal distribution $\Pr(R)$, write down the variable elimination expression with the elimination ordering $C, G, S$.
8. Give the complexities for each ordering. Which elimination ordering takes less computation?

**Solution:**

1. Here is a sketch of the graphical model:




It is based on $R$ and $S$ having marginal distributions that don't depend on any other variables, and $G$ having a conditional distribution based on both $R$ and $S$.

Using D-separation rules, $G$ is a node with converging arrows from $R$ and $S$. Therefore, $R$ and $S$ are independent iff $G$ is not in evidence. $R$ and $S$ are therefore independent but not conditionally independent when $G$ is observed.

2. From the independence of $R$ and $S$, this is simple (reduces to marginal probability):

$$P(S = 1|R = 1) = P(S = 1) = 0.5$$

3. Here, we can use Bayes' rule:

$$P(S = 1|G = 1) = \frac{P(G = 1|S = 1)P(S = 1)}{P(G = 1)}$$

Next, to find $P(G = 1|S = 1)$ and $P(G = 1)$, we can marginalize over all values $R = r$ and all $R = r$ as well as $S = s$ respectively:

$$= \frac{P(S = 1)\sum_{r=0}^{1} P(G = 1|S = 1, R = r)}{\sum_{r=0}^{1}\sum_{s=0}^{1} P(G = 1|S = s, R = r)P(R = r)P(S = s)}$$
$$= 0.8125$$

(after plugging in the values from the table)

4. For this part, we can use Bayes' rule with extra conditioning (everything is conditioned on $R$):

$$P(S = 1|G = 1, R = 1) = \frac{P(G = 1|S = 1, R = 1)P(S = 1|R = 1)}{P(G = 1|R = 1)}$$

The numerator here simplifies: the first term is 1 while the second term was computed in part 2. In the denominator, we can sum over $S = s$ to marginalize it out:

$$= \frac{0.5}{\sum_{s=0}^{1} P(G=1|R=1,S=s)P(S=s)} = \frac{0.5}{0.5 + 0.75 \cdot 0.5} = \frac{4}{7}$$

5. The explaining away effect shows that if a distribution of a variable (in this case $G$) depends on multiple explanatory factors ($R$ and $S$), then observing the variable along with one of the explanations makes the other explanation less likely. In this case, conditional on $G = 1$, $R$ and $S$ are negatively correlated. Specifically, $R$ "explains away" $S$ when it is observed.

   For a very intuitive explanation, consider the counterfactual world in which we're told that $G = 1$ and $R = 0$. If we know that it didn't rain, yet the grass is wet, then we'd know $S = 1$ with certainty. But then, conditional on $G = 1$, $R = 0$ and $S = 1$ are positively associated, which also means that the converse outcome $R = 1$ has to be negatively associated with $S = 1$.

6. Summing over the provided product expression for the joint distribution:

$$\Pr(R) = \sum_{s=0}^{1}\sum_{c=0}^{1}\sum_{g=0}^{1} \Pr(C,R,S,G)$$

$$= \sum_{s=0}^{1}\sum_{c=0}^{1}\sum_{g=0}^{1} \Pr(C)\Pr(R|C)\Pr(S|C)\Pr(G\,|\,R,S)$$

   Now rearranging sums in the desired order of elimination ($S$ goes last so that it gets eliminated first) gives:

$$= \sum_{c=0}^{1} \Pr(C)\Pr(R|C) \sum_{g=0}^{1}\sum_{s=0}^{1} \Pr(S|C)\Pr(G\,|\,R,S)$$

   this expression first marginalizes out $S$, then $G$, and finally $C$.

7. Summing over the provided product expression for the joint distribution:

$$\Pr(R) = \sum_{s=0}^{1}\sum_{c=0}^{1}\sum_{g=0}^{1} \Pr(C,R,S,G)$$

$$= \sum_{s=0}^{1}\sum_{c=0}^{1}\sum_{g=0}^{1} \Pr(C)\Pr(R|C)\Pr(S|C)\Pr(G\,|\,R,S)$$

   Now rearranging in the desired order:

$$= \sum_{s=0}^{1}\sum_{g=0}^{1} \Pr(G\,|\,R,S) \sum_{c=0}^{1} \Pr(C)\Pr(R|C)\Pr(S|C)$$

8. In part 6, the ordering $S, G, C$ requires computing $\Pr(G|R,S)$ and multiplying by $\Pr(S|C)$. This requires $(2^2) \cdot (2)$ values, so the sum-product can be computed in $2^4 = 16$ operations. Subsequent elimination stages are faster.

In part 7, the product $\Pr(C)\Pr(R|C)\Pr(S|C)$ requires $2^2$ values to store, so it takes $2^3$ operations to compute the sum-product. Subsequently, $\Pr(G|R,S)$ requires $2^2$ values to store, so it is not any slower. Thus, the second ordering of elimination takes less computation.

**Problem 2** (Policy and Value Iteration, 15 pts)

This question asks you to implement policy and value iteration in a simple environment called Gridworld. The "states" in Gridworld are represented by locations in a two-dimensional space. Here we show each state and its reward:

| | | | | |
|---|---|---|---|---|
| R=4 | R=0 | R=-10 | R=0 | R=20 |
| R=0 | R=0 | R=-50 | R=0 | R=0 |
| START R=0 | R=0 | R=-50 | R=0 | R=50 |
| R=0 | R=0 | R=-20 | R=0 | R=0 |

The set of actions is {N, S, E, W}, which corresponds to moving north (up), south (down), east (right), and west (left) on the grid. Taking an action in Gridworld does not always succeed with probability 1; instead the agent has probability 0.1 of "slipping" into a state on either side, but not backwards. For example, if the agent tries to move right from START, it succeeds with probability 0.8, but the agent may end up moving up or down with probability 0.1 each. Also, the agent cannot move off the edge of the grid, so moving left from START will keep the agent in the same state with probability 0.8, but also may slip up or down with probability 0.1 each. Lastly, the agent has no chance of slipping off the grid - so moving up from START results in a 0.9 chance of success with a 0.1 chance of moving right.

Also, the agent does not receive the reward of a state immediately upon entry, but instead only after it takes an action at that state. For example, if the agent moves right four times (deterministically, with no chance of slipping) the rewards would be +0, +0, -50, +0, and the agent would reside in the +50 state. Regardless of what action the agent takes here, the next reward would be +50.

Your job is to implement the following three methods in file `T6_P2.ipynb`. Please use the provided helper functions `get_reward` and `get_transition_prob` to implement your solution.

*Do not use any outside code. (You may still collaborate with others according to the standard collaboration policy in the syllabus.)*

*Embed all plots in your writeup.*

**Problem 2** (cont.)

**Important:** The state space is represented using integers, which range from 0 (the top left) to 19 (the bottom right). Therefore both the policy `pi` and the value function `V` are 1-dimensional arrays of length `num_states = 20`. Your policy and value iteration methods should only implement one update step of the iteration - they will be repeatedly called by the provided `learn_strategy` method to learn and display the optimal policy. You can change the number of iterations that your code is run and displayed by changing the `max_iter` and `print_every` parameters of the `learn_strategy` function calls at the end of the code.

Note that we are doing infinite-horizon planning to maximize the expected reward of the traveling agent. For parts 1-3, set discount factor $\gamma = 0.7$.

1a. Implement function `policy_evaluation`. Your solution should learn value function $V$, either using a closed-form expression or iteratively using convergence tolerance `theta = 0.0001` (i.e., if $V^{(t)}$ represents $V$ on the $t$-th iteration of your policy evaluation procedure, then if $|V^{(t+1)}[s] - V^{(t)}[s]| \leq \theta$ for all $s$, then terminate and return $V^{(t+1)}$.)

1b. Implement function `update_policy_iteration` to update the policy `pi` given a value function `V` using **one step** of policy iteration.

1c. Set `max_iter = 4`, `print_every = 1` to show the learned value function and the associated policy for the first 4 policy iterations. Do not modify the plotting code. Please fit all 4 plots onto one page of your writeup.

1d. Set `ct = 0.01` and increase `max_iter` such that the algorithm converges. Include a plot of the final learned value function and policy. How many iterations does it take to converge? Now try `ct = 0.001` and `ct = 0.0001`. How does this affect the number of iterations until convergence?

2a. Implement function `update_value_iteration`, which performs **one step** of value iteration to update `V`, `pi`.

2b. Set `max_iter = 4`, `print_every = 1` to show the learned value function and the associated policy for the first 4 value iterations. Do not modify the plotting code. Please fit all 4 plots onto one page of your writeup.

2c. Set `ct = 0.01` and increase `max_iter` such that the algorithm converges. Include a plot of the final learned value function and policy. How many iterations does it take to converge? Now try `ct = 0.001` and `ct = 0.0001`. How does this affect the number of iterations until convergence?

3 Compare and contrast the number of iterations, time per iteration, and overall runtime between policy iteration and value iteration. What do you notice?

4 Plot the learned policy with each of $\gamma \in (0.6, 0.7, 0.8, 0.9)$. Include all 4 plots in your writeup. Describe what you see and provide explanations for the differences in the observed policies. Also discuss the effect of gamma on the runtime for both policy and value iteration.

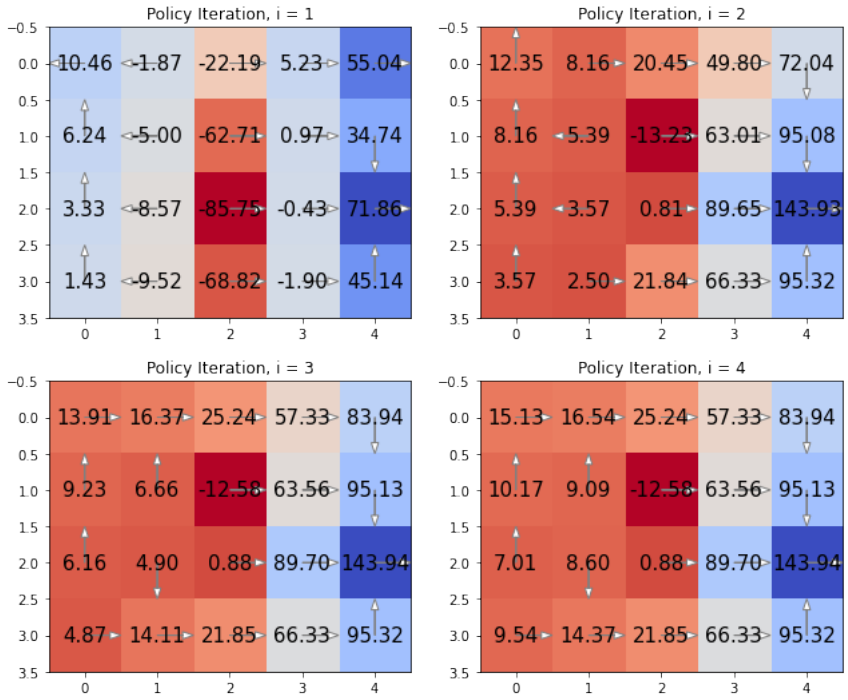
5 Now suppose that the game ends at any state with a positive reward, i.e. it immediately transitions you to a new state with zero reward that you cannot transition away from. What do you expect the optimal policy to look like, as a function of gamma? Numerical answers are not required, intuition is sufficient.

**Solution:**

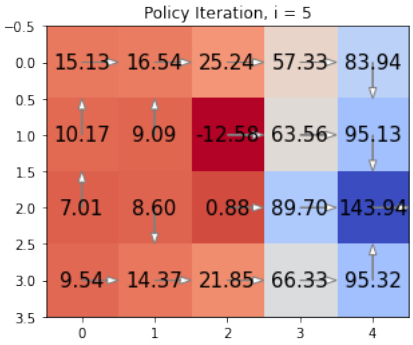
1a. implemented (see code file)

1b. implemented (see code file)

1c. Here are the 4 plots:




1d. Here is the final learned value function as well as policy:

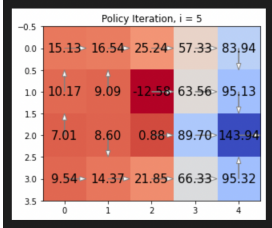



Convergence was achieved after just 5 steps with $ct = 0.01$. Also, remarkably, the policy iteration converged after 5 iterations for the cases when I set $ct = 0.001$ and $ct = 0.0001$. You can see the proof in the screenshots below:

```
print('Beginning policy iteration.')
learn_strategy(planning_type=POLICY_ITER, max_iter = 1000, print_every = 10, ct = 0.01, gamma = 0.7)
print('Policy iteration complete.\n')
```

✓ 1.3s

```
Beginning policy iteration.
```

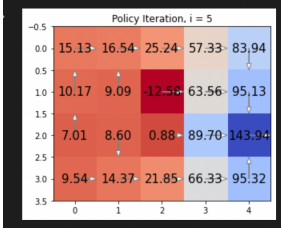



```
Converged at iteration 5
Policy iteration complete.
```

```
print('Beginning policy iteration.')
learn_strategy(planning_type=POLICY_ITER, max_iter = 1000, print_every = 10, ct = 0.001, gamma = 0.7)
print('Policy iteration complete.\n')
```

✓ 1.6s

```
Beginning policy iteration.
```

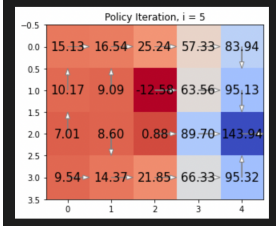



```
Converged at iteration 5
Policy iteration complete.
```

```
print('Beginning policy iteration.')
learn_strategy(planning_type=POLICY_ITER, max_iter = 1000, print_every = 10, ct = 0.0001, gamma = 0.7)
print('Policy iteration complete.\n')
```

✓ 1.4s

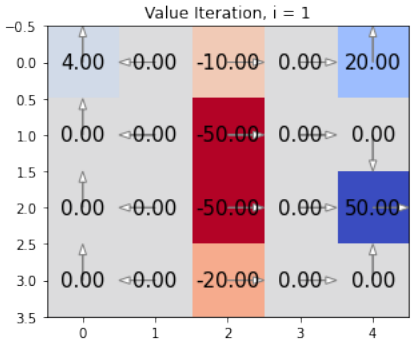
```
Beginning policy iteration.
```

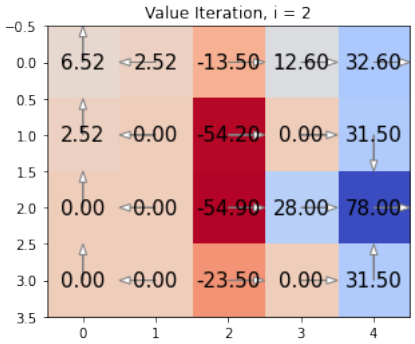


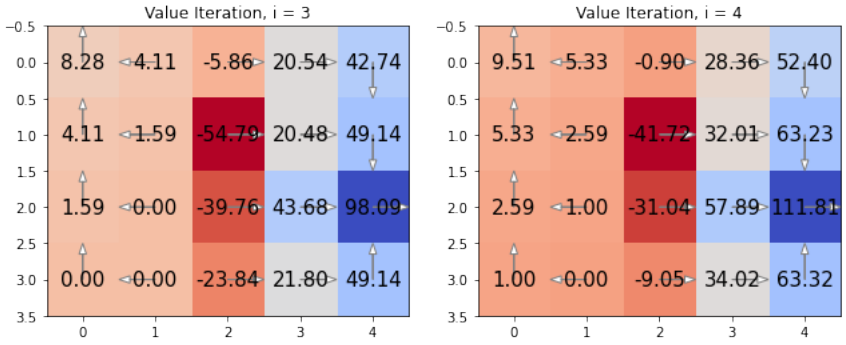

```
Converged at iteration 5
Policy iteration complete.
```

2a. implemented (see code file)

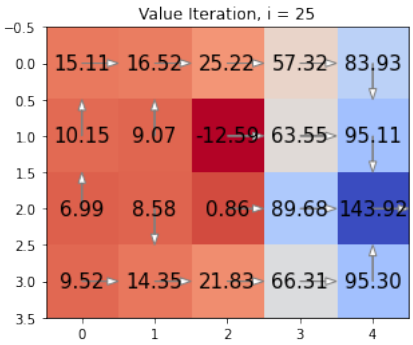
2b. See the value+policy plots for this setting below:






2c. This is the final plot for of the converged value and policy under policy iteration:



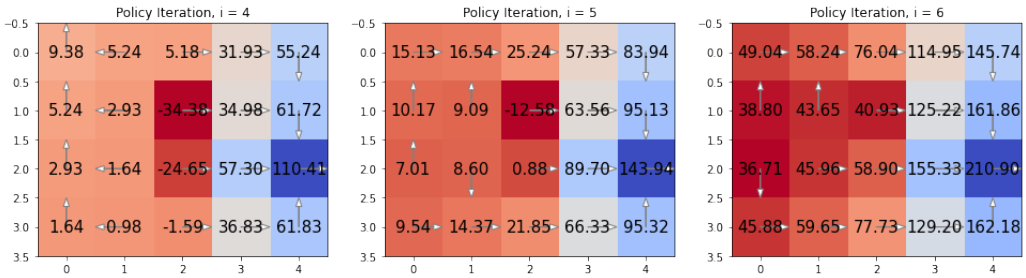

Decreasing the convergence tolerance $ct$ here increases the number of iterations required for convergence. Specifically, the algorithm converged in 25 iterations for $ct = 0.01$, in 31 iterations for $ct = 0.001$, and 38 iterations for $ct = 0.0001$.

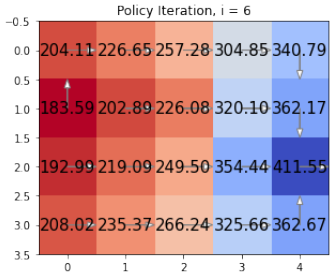
3. Here's a table showing the iterations, runtime, and average time per iteration for policy and value iteration at different ct values:

| algorithm | ct | iterations | runtime | time/iteration |
|---|---|---|---|---|
| policy | 0.01 | 5 | 0.9 sec | 0.18 sec |
| policy | 0.001 | 5 | 0.9 sec | 0.18 sec |
| policy | 0.0001 | 5 | 1.3 sec | 0.26 sec |
| value | 0.01 | 25 | 0.9 sec | 0.052 sec |
| value | 0.001 | 31 | 1.1 sec | 0.035 sec |
| value | 0.0001 | 38 | 1.3 sec | 0.034 sec |

Overall, the two methods seem comparable in total runtime. However, policy iteration requires fewer iterations, while value iterations requires less runtime per iteration.

4. Here are the plots for $\gamma = (0.6, 0.7, 0.8, 0.9)$ respectively:






For higher $\gamma$ values, policies skip immediate rewards in order to efficiently get to the state with $R = 50$. This makes sense: as we increase $\gamma$, we're implicitly placing a higher value on future rewards, so the agent becomes focused on long-term behavior over smaller but immediate rewards. In terms of runtime, here's a table with overall runtimes for both policy and value iterations with different $\gamma$ values. All results were obtained with $ct = 0.001$, the intermediate value.

| algorithm | $\gamma$ | runtime |
|---|---|---|
| policy | 0.6 | 0.8 sec |
| policy | 0.7 | 1.3 sec |
| policy | 0.8 | 1.7 sec |
| policy | 0.9 | 3.4 sec |
| value | 0.6 | 0.8 sec |
| value | 0.7 | 1.1 sec |
| value | 0.9 | 3.7 sec |

Increasing $\gamma$ seems to substantially and monotonically increase runtimes across both policy and value iterations. This is likely because the algorithm has to consider more future paths between states (since the future is valued more) and therefore requires more iterations to converge, as opposed to a low-$\gamma$ "greedy" agent that does what's locally optimal and then stops updating.

5. The optimal policy would be a path from the start to one of the 3 states with positive rewards. With high $\gamma$, the path would almost certainly end in the state with $R = 50$, as the agent is patient and this maximized reward. With lower $\gamma$, other positive-reward states would be picked. All paths would try to avoid incurring major losses (negative rewards) along the way, so they would likely avoid the two states with $-50$ rewards. For really low $\gamma$, the path would be simply "go from start to $R = 4$ field", with no loss in between.

**Problem 3** (Reinforcement Learning, 20 pts)

In 2013, the mobile game *Flappy Bird* took the world by storm. You'll be developing a Q-learning agent to play a similar game, *Swingy Monkey* (See Figure 1a). 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, the gravity varies from game to game, 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. However, your objective is to build an agent that *learns* to play on its own.

You will need to install the `pygame` module (http://www.pygame.org/wiki/GettingStarted).

**Task:** Your task is to use Q-learning to find a policy for the monkey that can navigate the trees. The implementation of the game itself is in file `SwingyMonkey.py`, along with a few files in the `res/` directory. A file called `stub.py` is the starter code for setting up your learner that interacts with the game. This is the only file you need to modify (but to speed up testing, you can comment out the animation rendering code in `SwingyMonkey.py`). You can watch a YouTube video of the staff Q-Learner playing the game at http://youtu.be/l4QjPr1uCac. It figures out a reasonable policy in a few dozen iterations. You'll be responsible for implementing the Python function `action_callback`. The action callback will take in a dictionary that describes the current state of the game and return an action for 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> }}
```

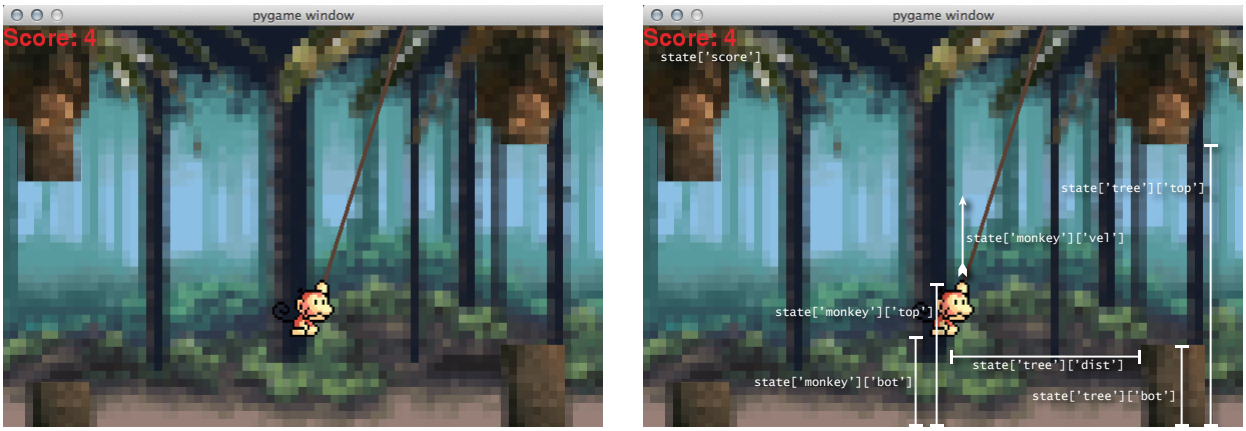
All of the units here (except score) will be in screen pixels. Figure 1b shows these graphically. Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

**Requirements**

*Code*: First, you should implement Q-learning with an $\epsilon$-greedy policy yourself. You can increase the performance by trying out different parameters for the learning rate $\alpha$, discount rate $\gamma$, and exploration rate $\epsilon$. *Do not use outside RL code for this assignment.* Second, you should use a method of your choice to further improve the performance. This could be inferring gravity at each epoch (the gravity varies from game to game), updating the reward function, trying decaying epsilon greedy functions, changing the features in the state space, and more. One of our staff solutions got scores over 800 before the 100th epoch, but you are only expected to reach scores over 50 before the 100th epoch. **Make sure to turn in your code!**

*Evaluation*: In 1-2 paragraphs, explain how your agent performed and what decisions you made and why. Make sure to provide evidence where necessary to explain your decisions. You must include in your write up at least one plot or table that details the performances of parameters tried (i.e. plots of score vs. epoch number for different parameters).

*Note*: Note that you can simply discretize the state and action spaces and run the Q-learning algorithm. There is no need to use complex models such as neural networks to solve this problem, but you may do so as a fun exercise.




(a) SwingyMonkey Screenshot

(b) SwingyMonkey State

Figure 1: (a) Screenshot of the Swingy Monkey game. (b) Interpretations of various pieces of the state dictionary.

**Solution:**

I initially implemented the simple Q-learning agent without any modifications. It performed poorly (getting a maximum score of 5) for $\epsilon > 0.1$, $\alpha > 0.5$, and $\gamma < 0.5$. I was able to improve its performance by choosing better parameter values, finally settling on $\epsilon = 0.02$ to discourage random actions late in the game (when the agent has already learned a sensible policy), $\alpha = 0.1$ to prevent hasty updates that completely reverse a learned policy (for example when gravity changes from game to game), and $\gamma = 0.9$ to make the agent care about future payoffs, which produces more consistent results in later epochs (after epoch 60).

The biggest impact came from changing $\epsilon$, likely because the game is binary: you keep getting rewards by jumping successfully and then instantly lose when the agent makes a mistake. Random actions can therefore have a very high cost if they break a chain of decisions that could have led to dozens if not hundreds of points. Also, note that with the right parameter values, the simple Q-learning agent was able to meet the benchmark, exceeding 50 before the 100th iteration in every run and getting a high score of 138.

To further improve the agent, I decided to implement a decaying $\epsilon$ value. This means that with every action, the algorithm moves from $\epsilon$-greedy slightly closer to greedy. Specifically, the model is as follows:

$$\epsilon_t = \epsilon_0 \cdot \delta^t$$

where $\epsilon_t$ is the epsilon parameter after action $t$, $\epsilon_0$ is the initial value of $\epsilon$, and $\delta \in (0, 1)$ is the decay parameter. The justifications for this are twofold: (1) decaying epsilon ensures that the agent takes very few random actions in late game. This is great since the agent has already learned a useful policy. (2) decaying epsilon allows me to set a high staring value $\epsilon_0$. This encourages exploration at the beginning of the game, ensuring faster convergence to an optimal policy. Below, you can find a table of results for parameter tuning, where I try different values of $\epsilon_0$ and $\delta$ to see how they impact the median, mean, and maximum score across 100 epochs. This measures consistency, average performance, and optimal performance.

| $\epsilon_0$ | $\delta$ | median | mean | max |
|---|---|---|---|---|
| 0.1 | 0.9 | 2.0 | 37.92 | 906 |
| 0.1 | 0.8 | 2.0 | 84.02 | 1020 |
| 0.1 | 0.7 | 1.0 | 45.03 | 733 |
| 0.2 | 0.9 | 3.0 | 71.2 | 792 |
| 0.2 | 0.8 | 1.0 | 51.28 | 1136 |
| 0.2 | 0.7 | 2.0 | 59.77 | 707 |
| 0.5 | 0.9 | 1.0 | 25.87 | 722 |
| 0.5 | 0.9 | 1.0 | 42.52 | 681 |
| 0.5 | 0.9 | 2.0 | 49.09 | 451 |

From this, it is clear that decaying $\epsilon$ massively improves the agent's performance. Also, the optimal parameter values that strike a compromise between maximum performance and consistency are likely close to $\epsilon_0 = 0.1$ and $\delta = 0.8$

My code is below:

```
# Imports.
import numpy as np
import numpy.random as npr
import pygame as pg

# uncomment this for animation
#from SwingyMonkey import SwingyMonkey

# uncomment this for no animation
from SwingyMonkeyNoAnimation import SwingyMonkey


X_BINSIZE = 200
Y_BINSIZE = 100
X_SCREEN = 1400
Y_SCREEN = 900


class Learner(object):
    """
    This agent jumps randomly.
    """

    def __init__(self):
        self.last_state = None
        self.last_action = None
        self.last_reward = None

        # We initialize our Q-value grid that has an entry for each action and state.
        # (action, rel_x, rel_y)
        self.Q = np.zeros((2, X_SCREEN // X_BINSIZE, Y_SCREEN // Y_BINSIZE))
        self.epsilon = 0.05

    def reset(self):
        self.last_state = None
        self.last_action = None
```

```
        self.last_reward = None

    def discretize_state(self, state):
        """
        Discretize the position space to produce binned features.
        rel_x = the binned relative horizontal distance between the monkey and the tree
        rel_y = the binned relative vertical distance between the monkey and the tree
        """

        rel_x = int((state["tree"]["dist"]) // X_BINSIZE)
        rel_y = int((state["tree"]["top"] - state["monkey"]["top"]) // Y_BINSIZE)
        return (rel_x, rel_y)

    def action_callback(self, state):
        """
        Implement this function to learn things and take actions.
        Return 0 if you don't want to jump and 1 if you do.
        """

        # TODO (currently monkey just jumps around randomly)
        # 1. Discretize 'state' to get your transformed 'current state' features.
        # 2. Perform the Q-Learning update using 'current state' and the 'last state'.
        # 3. Choose the next action using an epsilon-greedy policy.

        new_state = state

        alpha = 0.1
        gamma = 0.9
        epsilon = self.epsilon

        current_state = self.discretize_state(state)
        cx = current_state[0]
        cy = current_state[1]
        if (self.last_state is None):
            if npr.rand() >= epsilon:
                new_action = np.argmax([self.Q[0,cx,cy], self.Q[1,cx,cy]])
            else:
                new_action = int((npr.rand() > 0.5))
        else:
            last_state = self.discretize_state(self.last_state)
            lx = last_state[0]
            ly = last_state[1]
            self.Q[int(self.last_action),lx,ly] += alpha * (self.last_reward + gamma*np.ma
            if npr.rand() >= epsilon:
                new_action = np.argmax([self.Q[0,cx,cy], self.Q[1,cx,cy]])
            else:
                new_action = int((npr.rand() > 0.5))

        self.last_action = new_action
        self.last_state = new_state
```

```
        self.epsilon = self.epsilon * 0.8

        return self.last_action

    def reward_callback(self, reward):
        """This gets called so you can see what reward you get."""

        self.last_reward = reward


def run_games(learner, hist, iters=100, t_len=100):
    """
    Driver function to simulate learning by having the agent play a sequence of games.
    """
    for ii in range(iters):
        # Make a new monkey object.
        swing = SwingyMonkey(sound=False,  # Don't play sounds.
                             text="Epoch %d" % (ii),  # Display the epoch on screen.
                             tick_length=t_len,  # Make game ticks super fast.
                             action_callback=learner.action_callback,
                             reward_callback=learner.reward_callback)

        # Loop until you hit something.
        while swing.game_loop():
            pass

        # Save score history.
        hist.append(swing.score)

        # Reset the state of the learner.
        learner.reset()
    pg.quit()
    return


if __name__ == '__main__':
    # Select agent.
    agent = Learner()

    # Empty list to save history.
    hist = []

    # Run games. You can update t_len to be smaller to run it faster.
    run_games(agent, hist, 100, 100)
    print(hist)

    # Save history.
    np.save('hist', np.array(hist))
```

## Name

Matej Cerman

## Collaborators and Resources

Whom did you work with, and did you use any resources beyond cs181-textbook and your notes?

Worked with Soren Choi and Arpit Bhate.

## Calibration

12-15 hours because of the insane amount of busywork in problem 2. Please don't do this anymore.

Problem 3 was so much fun though! This is what the class should be! So cool!