Open in Colab

ENGS 108 Fall 2020 Assignment 6

Due November 9, 2020 at 11:59PM on Canvas

Instructors: George Cybenko

TAs: Chase Yakaboski

Rules and Requirements

1. You are only allowed to use Python packages that are explicity imported in the assignment notebook or are standard (bultin) python libraries like random, os, sys, etc, (Standard Bultin Python libraries will have a Python.org documentation). For this assignment you may use:
   - numpy
   - pandas
   - scikit-learn
   - matplotlib
   - tensorflow
2. All code must be fit into the designated code or text blocks in the assignment notebook. They are indentified by a **TODO** qualifier.
3. For analytical questions that don't require code, type your answer cleanly in Markdown. For help, see the Google Colab Markdown Guide.

In [ ]:
```
''' Import Statements '''
from collections import defaultdict
import copy
import itertools

import random
# Don't mess with this (gives reproducible results)
random.seed(444)
```

Problem 1: Reinforcement Learning

In this problem we will play a game of cops and robbers. The game is played on a fixed undirected, simple, and finite graph G. There are two players, a cop and a robber. It is the goal of the cop to catch the robber in as few moves as possible.

The graph G has the following properties:

- A total of m nodes.
- It contains a single n node cycle, where $n \leq m$, and random additional edges to make the graph connected.

The game starts, with the cop taking their choice of vertex in G and then the robber selects a random vertex in G that is not occupied by the cop. At every point in the game both players know the positions of each other, and in this version of the problem we will say that the robber is drunk (i.e. they will randomly choose there next that instead of employing a policy).

The availabe actions of the cop and associated reward function is:

- Move to a node not connected to their present node (and the cop stays in the current position): -5.
- Move to an adjacent node (including staying at current node): -1.
- Move to the node occupied by robber: +100. >

> **Part 1** Building a Graph Class.
>
> > **(a)** Using the provided skeleton build a general graph class for this problem for n nodes. You are expected to implement *add_edge*, *make_random_graph*, *check_connected*.

In [ ]:
```
class Graph:
  """ Our graph class.
  Args:
    n: Number of nodes in our cycle.
    m: Total number of nodes in graph, where n >= m.
  """
  def __init__(self, n, m):
    self.n = n
    self.m = m
    # A default dict is just a dict that won't raise a KeyError, it instead fills
    # the unknown key with a default, in our case an empty list.
    self.G = defaultdict(list)

  def add_edge(self, u, v):
    """
    Make a function that will add an edge to the graph.
    """
    #TODO: Implement.


  def make_random_graph(self):
    """ First make a cycle of given length and then add random additional edges
    in such a way that the final graph will be connected.
    """
    #TODO: Make n length cycle first

    #TODO: Add additional nodes until random graph is connected
    while True:
      _graph = copy.copy(self)
      #TODO: Use the graph copy to add random nodes/edges
      #TODO: Check if the new graph is connected.
      if self.check_connected(_graph.G):
        # If it is set the current graph's adjacency matrix equal to the copy's.
        self.graph = _graph.G
        # Return (i.e. break the loop)
        return True

  def check_connected(self, G):
    """ Perform a Depth First Search.
    """
    start_node = random.choice(list(G.keys()))
    return dfs(G, {}, start_node, start_node)

def dfs(G, visited, u, parent):
  """ Implement your own depth first search. Many online resources for this.
  Args:
    G: Is the graph (adjency matrix) we want to test.
    visited: Is a dictionary that keeps track of the nodes you've visited.
    u: a starting node.
    parent: The parent node for DFS so that you can cancel recursion if you complete the cycle.
  Returns:
    True: If all nodes have been visited at least once.
    False: Otherwise.
  """
  #TODO: Implement. Hint: Make sure to keep track of the parent node for a DFS search,
  # as a cycle leads to an infinite recursion, so you have to make sure you break the cycle.
```

> > **(b)** Test out your graph class with a cops and robbers graph of $n = 5$ and $m = 10$.

In [ ]:
```
#TODO: Your code goes here.
```

> **Part 2** Understanding the state space, i.e. the Game.
>
> > **(a)** Given the graph class you've created in Part 1. Develop a Cops and Robbers game class. Use the skeleton below to implement the following functions first: *get_successors*, *terminal_test*, *result*.

In [ ]:
```
class CopsAndRobbers:
  def __init__(self, graph, start_state, rewards_table=None):
    """ This is the cops and robbers game class.
    Args:
      start_state: The starting state of the cop position and robber position in
        the graph. Should be a tuple of form (cop_pos, rob_pos).
      G: The graph adjacency matrix
      state: current state in the game.
      reward_table: The (state, actions) reward dictionary that you will eventually implement.
    """
    self.graph = graph
    self.G = graph.G
    self.state = start_state
    self.rewards_table = rewards_table

  def terminal_test(self, state):
    #TODO: Implement.

  def get_successors(self, state):
    """ Return a list of successor states that can be reached from the current state.
    Hint: Remember only the cop can choose their action.
    """
    #TODO: Implement.

  def result(self, state, next_state):
    """ This function should return the state after the cop has made their move,
    and the drunk robber has moved accordingly.
    Args:
      state: Current state of (cop, rob).
      next_state: The state after the cop has moved (next_cop, rob). Calculated from get_successors.
    """
    # TODO: Implement.

  def utility(self, state, action):
    return self.rewards_table[(state, action)]
```

> > **(b)** In reinforcement learning we are often interested in calculating a rewards table that has possible states as its rows and possible actions as its columns and filled in with the associated reward given the Q(state, action) pair. Calculate the rewards table for any given graph. *Hint: This should be an m^2 x m matrix or a dictionary with m^3 keys such that the keys are (state, action) tuples.*

In [ ]:
```
def calculateRewardsTable(graph):
  """ Make a rewards table dictionary of the from table[(state, action)] = reward.
  """
  #TODO: implement
  return table
```

> > **(c)** Now that we have our reward table, try to solve the problem in a brute-force manner (without reinforcement learning). I.e. try to reach the terminal state, or find get a reward of 100.

In [ ]:
```
epochs = 0
rewards = []
penalties = 0
#TODO: Instantiate your graph
#TODO: Build your rewards table
#TODO: Instantiate your game.

# Simulation loop
while True:
  #TODO: Get the next state, check for terminal state, and get reward
  # Make sure to break if you reach the terminal state.
```

> **Part 3** Q-learning.
>
> Up to this point we have build a graph class, built a game class, ran a brute-force simulation of an agent traversing the space randomly, and now we will dive into Q-learning in the hopes of maximizing the rewards and efficiency of capturing the drunk robber.
>
> > **(a)** Using the skeleton from the brute force method, implement a training loop to learn a Q-table for a given graph and game.

In [ ]:
```
#TODO: Your code goes here.
```

> > **(b)** Evalute your new Q-learning agent over a 100 epochs, by choosing your actions based on the argmax of the Q-table caluculated in (a) and report the average number of penalities, average time, and average number of steps it took to find the robber with your new Q-learning strategy.

In [ ]:
```
#TODO: Your code goes here.
```

> > **(c)** Compare your results with the brute-force method used in Part 2 and comment on the improvement. For instance, try varying graph configurations and look for any signs of improvement in certian instances.

In [ ]:
```
#TODO: Your code goes here.
```

> **Bonus** Check that the learned policy satsifies the Bellman Inequality, i.e is the computed solution the actual optimal policy?

In [ ]:
```
#TODO: Your code goes here.
```