MDPs and Q-learning On "Ice" (60 points possible)

In this assignment, we'll revisit Markov Decision Processes while also trying out Q-Learning, the reinforcement learning approach that associates utilities with attempting actions in states. The problem that we're attempting to solve is the following:

1. There is a grid of spaces in a rectangle. Each space can contain a pit (negative reward), gold (positive reward), or nothing.
2. The rectangle is effectively surrounded by walls, so anything that would move you outside the rectangle, instead moves you to the edge of the rectangle.
3. The floor is icy. Any attempt to move in a cardinal direction results in moving a somewhat random number of spaces in that direction. The exact probabilities of moving each number of spaces are given in the problem description. (If you slide too far, see rule #2.)
4. Landing on a pit or gold effectively "ends the run," for both a Q learner and an agent later trying out the policy. It's game over. (To simulate this during Q learning, set all Q values for the space to equal its reward, then start over from a random space.) Note that it's still possible to slide past a pit or gold - this doesn't end the run.

A sample input looks like this:

```
sampleMDP = """0.7 0.2 0.1
- - P - -
- - G P -
- - P - -
- - - - -"""
```

The first line says that the probabilities of moving one, two, or three spaces in the direction of movement are 0.7, 0.2, and 0.1. The rest is a map of the environment, where a dash is an empty space, P is a pit, and G is gold.

Your job is to finish the code below for mdp_solve() and q_solve(). These take a problem description like the one pictured above, and return a policy giving the recommended action to take in each empty square (U=up, R=right, D=down, L=left).

1, 17 points) mdp_solve() should use value iteration and the Bellman equation. ITERATIONS will refer to the number of complete passes you perform over all states. You can initialize the utilities to the rewards of each state. Don't update the rewards spaces from their initial rewards; since they end the trial, they have no future utility. Don't update utilities in-place as you iterate through them, but create a fresh array of utilities with each pass, in order to avoid biasing moves in the directions that have already been updated.

2, 24 points) q_solve() will run ITERATIONS trials in which a learner starts in a random empty square and moves until it hits a pit or gold, in which case, the trial is over. (If it was randomly dropped into gold or a pit, the trial is immediately over.) The learner moves by deciding randomly whether to choose a random direction (with probability EXPLORE_PROB) or move according to the best Q-value of its current square (otherwise). Simulate the results of the move on slippery ice to determine where the learner ended up - then apply the Q-learning equation given in lecture and the textbook. (There are multiple Q-learning variants out there, so try to use the equations and practices described in lecture instead of using other sources, to avoid confusion.)

The fact that a trial ends immediately on finding gold or a pit means that we want to handle those spaces in a special way. Normally Q values are updated on moving to the next state, but we won't see any next state in these cases. So, to handle this, when the agent discovers one of these rewards, set all the Q values for that space to the associated reward before quitting the trial. So, for example, if gold is worth 100 and it's discovered in square x, Q(x,UP) = 100, Q(x,RIGHT) = 100, Q(x, DOWN) = 100, and Q(x, LEFT) = 100. There's no need to apply the rest of the Q update equation when the trial is ending, because that's all about future rewards, and there's no future when the trial is ending. But now the spaces that can reach that space will evaluate themselves appropriately. (Before being "discovered," the square should have no utility.)

You should use the GOLD_REWARD, PIT_REWARD, LEARNING_RATE, and DISCOUNT_FACTOR constants at the top of the code box below.

Q-learning involves a lot of randomness and some arbitrary decisions when breaking ties, so two implementations can both be correct but recommend slightly different policies in the end, even if they have the same starting random seed. While we provide some helpful premade maps below, your main guide for debugging will be common sense in deciding whether the policy created by your agent makes sense -- ie, agents following the policy will get gold without taking unnecessary risks.

```
import numpy as np


""" "MDPs on Ice - Assignment 5"""

import random
import copy

GOLD_REWARD = 250.0
PIT_REWARD = -150.0
DISCOUNT_FACTOR = 0.8
EXPLORE_PROB = 0.2 # for Q-learning
LEARNING_RATE = 0.01
ITERATIONS = 20000
MAX_MOVES = 1000
ACTIONS = 4
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3
MOVES = ['U', 'R', 'D', 'L']

# Fixed random number generator seed for result reproducibility --
# don't use a random number generator besides this to match sol
random.seed(340)

class Problem:
    """Represents the physical space, transition probabilities, reward locations, and approach

    ...in short, the info in the problem string

    Attributes:
        move_probs (List[float]):  probabilities of going 1,2,3 spaces
        map (List[List(string)]]:  "-" (safe, empty space), "G" (gold), "P" (pit)

    String format consumed looks like
    0.7 0.2 0.1   [probability of going 1, 2, 3 spaces]
    - - - - - - P - - - -   [space-delimited map rows]
    - - G - - - - - P - -   [G is gold, P is pit]

    You can assume the maps are rectangular, although this isn't enforced
    by this constructor.
    """

    def __init__(self, probstring):
        """ Consume string formatted as above"""
        self.map = []
        for i, line in enumerate(probstring.splitlines()):
            if i == 0:
                self.move_probs = [float(s) for s in line.split()]
            else:
                self.map.append(line.split())

    def solve(self, iterations, use_q):
        """ Wrapper for MDP and Q solvers.

        Args:
            iterations (int):  Number of iterations (but these work differently for the two solvers)
            use_q (bool):  False means use MDP value iteration, true means use Q-learning
        Returns:
            A Policy, in either case (what to do in each square; see class below)
        """

        if use_q:
            return q_solve(self, iterations)
        return mdp_solve(self, iterations)

class Policy:
    """ Abstraction on the best action to perform in each state.

    This is a string list-of-lists map similar to the problem input, but a character gives the best
    action to take in each non-reward square (see MOVES constant at top of file).
    """

    def __init__(self, problem):
        """Args:

        problem (Problem):  The MDP problem this is a policy for
        """
        self.best_actions = copy.deepcopy(problem.map)

    def __str__(self):
        """Join the characters in the policy into one big space-separated, multline string"""
        return '\n{}\n'.format('\n'.join([' '.join(row) for row in self.best_actions]))

def roll_steps(move_probs, row, col, move, rows, cols):
    """Calculates the new coordinates that result from a move.

    Includes the "roll of the dice" for transition probabilities and checking arena boundaries.

    Helper for try_policy and q_solve - probably useful in your Q-learning implementation.

    Args:
        move_probs (List[float]):  Transition probabilities for the ice (from problem)
        row, col (int, int):  location of agent before moving
        move (string):  The direction of move as a MOVES character (not an int constant!)
        rows, cols (int, int):  number of rows and columns in the map

    Returns:
        new_row, new_col (int, int):  The new row and column after moving
    """
    displacement = 1
    total_prob = 0
    move_sample = random.random()
    for p, prob in enumerate(move_probs):
        total_prob += prob
        if move_sample <= total_prob:
            displacement = p+1
            break
    # Handle "slipping" into edge of map
    new_row = row
    new_col = col
    if not isinstance(move, str):
        print("Warning: roll_steps wants str for move, got a different type")
    if move == "U":
        new_row -= displacement
        if new_row < 0:
            new_row = 0
    elif move == "R":
        new_col += displacement
        if new_col >= cols:
            new_col = cols-1
    elif move == "D":
        new_row += displacement
        if new_row >= rows:
            new_row = rows-1
    elif move == "L":
        new_col -= displacement
        if new_col < 0:
            new_col = 0
    return new_row, new_col


def try_policy(policy, problem, iterations):
    """Returns average utility per move of the policy.

    Average utility is as measured by "iterations" random drops of an agent onto empty
    spaces, running until gold, pit, or time limit MAX_MOVES is reached.

    Doesn't necessarily play a role in your code, but you can try policies this
    way

    Args:
        policy (Policy):  the policy the agent is following
        problem (Problem):  the environment description
        iterations (int):  the number of random trials to run
    """
    total_utility = 0
    total_moves = 0
    for _ in range(iterations):
        # Resample until we have an empty starting square
        while True:
            row = random.randrange(0, len(problem.map))
            col = random.randrange(0, len(problem.map[0]))
            if problem.map[row][col] == "-":
                break
        for moves in range(MAX_MOVES):
            total_moves += 1
            policy_rec = policy.best_actions[row][col]
            # Take the move - roll to see how far we go, bump into map edges as necessary
            row, col = roll_steps(problem.move_probs, row, col, policy_rec, \
                                  len(problem.map), len(problem.map[0]))
            if problem.map[row][col] == "G":
```

```
                total_utility += GOLD_REWARD
                break
            if problem.map[row][col] == "P":
                total_utility += PIT_REWARD
                break
        return total_utility / total_moves

    def space_utility(space):
        utility = 0
        if space == 'P':
            utility = PIT_REWARD
        elif space == 'G':
            utility = GOLD_REWARD
        return utility

    # return the space within the bounds of the board (new_row, new_col) as coords
    def get_space(problem, row, col, row_change, col_change, dist):
        return min(max(0, row + (row_change * dist)), len(problem.map) - 1), min(max(0, col + (col_change * dist)), len(problem.map[row]) - 1)

    def mdp_solve(problem, iterations):
        """ Perform value iteration for the given number of iterations on the MDP problem.

        Here, the squares with rewards can be initialized to the reward values, since value iteration
        assumes complete knowledge of the environment and its rewards.

        Args:
            problem (Problem):  description of the environment
            iterations (int):  number of complete passes over the utilities
        Returns:
            a Policy (though you may design this to return utilities as a second return value)
        """
        # TODO calculate the policy
        policy = Policy(problem=problem)
        # initialize utility map
        utility_map = []
        # print(len(utility_map))
        for i, row in enumerate(problem.map):
            for j, space in enumerate(row):
                if(j == 0):
                    utility_map.append([])
                utility_map[i].append(space_utility(space))

        # print(problem.map)
        # print(utility_map)

        for iter in range(iterations):
            new_utility_map = []
            # run the bellman equation to update the value property for each space
            for i in range(len(utility_map)):
                # print("i: " + str(i))
                # print(utility_map)
                # print(utility_map[i])
                for j in range(len(utility_map[i])):
                    if(j == 0):
                        new_utility_map.append([])
                    max_util = 0
                    for move in MOVES:
                        # changes for 'U'
                        row_change = -1
                        col_change = 0
                        if move == 'R':
                            row_change = 0
                            col_change = 1
                        elif move == 'D':
                            row_change = 1
                        elif move == 'L':
                            row_change = 0
                            col_change = -1

                        # calculate the bellman equation sum for each distance that the player can move
                        sum = 0
                        for k,prob in enumerate(problem.move_probs):
                            new_row, new_col = get_space(problem, i, j, row_change, col_change, k + 1)
                            # new_space = problem.map[new_row][new_col]
                            sum += prob*(DISCOUNT_FACTOR * utility_map[new_row][new_col] + space_utility(problem.map[i][j]))
                        if sum > max_util:
                            policy.best_actions[i][j] = move
                            max_util = sum
                    new_utility_map[i].append(max_util)
            utility_map = new_utility_map


        return policy, utility_map

    def q_solve(problem, iterations):
        """q_solve:  Use Q-learning to find a good policy on an MDP problem.

        Each iteration corresponds to a random drop of the agent onto the map, followed by moving
        the agent until a reward is reached or MAX_MOVES moves have been made.  When an agent
        is sitting on a reward, update the utility of each move from the space to the reward value
        and end the iteration.  (For simplicity, the agent also does this if just dropped there.)
        The agent does not "know" reward locations in its Q-values before encountering the space
        and "discovering" the reward.

        Note that texts differ on when to pay attention to this reward - this code follows the
        convention of scoring rewards of the space you are moving *from*, plus discounted best q-value
        of where you landed.

        Assume epsilon-greedy exploration.  Leave reward letters as-is in the policy,
        to make it more readable.

        Args:
            problem (Problem):  The environment
            iterations (int):  The number of runs from random start to reward encounter
        Returns:
            A Policy for the map
        """
        # TODO
        policy = Policy(problem=problem)
        rows = len(problem.map)
        cols = len(problem.map[0])
        # store the Q(a, s) for each square shape = (row, col, 4)
        Q = np.zeros(shape=(rows, cols, len(MOVES)))

        # create the rewards map
        rewards = np.zeros(shape=(rows, cols))
```

```
        for i in range(rows):
            for j in range(cols):
                rewards[i][j] = space_utility(problem.map[i][j])

        for iter in range(iterations):
            # for each iteration, run an iteration of q learning
            # drop the agent onto a random space on the map
            row = random.randint(0, rows - 1)
            col = random.randint(0, cols - 1)
            for i in range(MAX_MOVES):
                # check if the current space is a PIT or GOLD
                # insert code here
                if problem.map[row][col] == 'G' or problem.map[row][col] == 'P':
                    for j in range(len(MOVES)):
                        Q[row][col][j] = space_utility(problem.map[row][col])
                    break
                # until it reaches MAX_MOVES (or lands on gold or pit) keep moving it
                if random.random() <= EXPLORE_PROB:
                    # pick a random direction to explore
                    move = random.randint(0, 3)
                    # get the new space given the chosen direction
                    new_row, new_col = roll_steps(move_probs=problem.move_probs, row=row, col=col, move=MOVES[move], rows=rows, cols=cols)
                    # Q[row][col][move] += LEARNING_RATE*(space_utility(problem.map[row][col]) + DISCOUNT_FACTOR*max(Q[new_row][new_col]) - Q[row][col][move])
                    Q[row][col][move] = new_q(rewards=rewards, utilities=Q, r=row, c=col, new_r= new_row, new_c=new_col, movenum=move)

                    row = new_row
                    col = new_col
                else:
                    # pick the best action for the given space
                    move = np.argmax(Q[row][col])
                    new_row, new_col = roll_steps(problem.move_probs, row, col, MOVES[move], rows, cols)
                    # Q[row][col][move] += LEARNING_RATE*(space_utility(problem.map[row][col]) + DISCOUNT_FACTOR*max(Q[new_row][new_col]) - Q[row][col][move])
                    Q[row][col][move] = new_q(rewards=rewards, utilities=Q, r=row, c=col, new_r= new_row, new_c=new_col, movenum=move)
                    row = new_row
                    col = new_col
        # print(Q)

        # populate the best moves for each position
        for i in range(rows):
            for j in range(cols):
                policy.best_actions[i][j] = MOVES[np.argmax(Q[i][j])]

        return policy

    def new_q(rewards, utilities, r, c, new_r, new_c, movenum):
        """ Q-learning function.  Returns the new Q-value for space (r,c).
        It's recommended you code and test this before doing the overall Q-learning.

        Should use the LEARNING_RATE and DISCOUNT_FACTOR.

        Args:
            rewards (List[List[float]]):  Reward amounts built into the problem map (indexed [r][c])
            utilities (List[List[List[float]]]):  The Q-values for each action from each space.
                                                  (Indexed as [row][col][move])
            r, c (int, int):  Row and column of our location before move
            new_r, new_c (int, int):  Row and column of our location after move
            movenum (int):  Integer index into the Q-values, corresponding to constants UP etc
        Returns:
            float - the new Q-value for the space we moved from
        """
        # TODO
        new_q = utilities[r][c][movenum] + LEARNING_RATE * (rewards[r][c] + DISCOUNT_FACTOR * max(utilities[new_r][new_c]) - utilities[r][c][movenum])
        return new_q


    deterministic_test = """1.0
    - - P - -
    - - G P -
    - - P - -
    - - - - -"""


    # Notice that we counterintuitively are most likely to go 2 spaces here
    very_slippy_test = """0.2 0.7 0.1
    - - P - -
    - - G P -
    - - P - -
    - - - - -"""


    big_test = """0.6 0.3 0.1
    - P - G - P - - G -
    P G - P - - - P - -
    P P - P P - P - P -
    P - - P P - - - - P
    - - - - - - - - P G"""


    # MDP value iteration tests
    p, u = Problem(deterministic_test).solve(ITERATIONS, False)
    print(p)
    print(u)
    # print(p.best_actions)
```

```
R D D D D
R R L L L
U U U U U
U U U U U

[[355.5555555555556, 444.44444444444446, 405.55555555555554, 324.44444444444446, 259.5555555555556], [444.44444444444446, 555.5555555555555, 694.4444444444445, 405.55555555555554, 324
```

```
p, u = Problem(sampleMDP).solve(ITERATIONS, False)
print(p)
print(u)
```

```
D D D L D
R R L L L
U U U U U
U U U L U

[[207.72758935374807, 243.86348225368286, 177.0497361881302, 154.78421657424204, 163.99373373654106], [285.36920834977286, 335.01148832783446, 506.09504346753283, 209.8445991422537, 2
```

```
p, u = Problem(very_slippy_test).solve(ITERATIONS, False)
print(p)
print(u)
```

```
D D D R D
R L L L L
D D D R D
U U U R U

[[176.5115520543013, 141.20924164344106, 5.499167388424542, 135.83226925188637, 169.79033656485797], [336.540040056473, 269.2320320451784, 508.4627507633712, 109.04718127195716, 323.7
```

```
p, u = Problem(big_test).solve(ITERATIONS, False)
print(p)
print(u)
```

```
R R R U L L R R U L
R R U U U L U U U U
U U U U U U U R U D
R U U U U R D D U D
R U U U R R R R R R

[[607.3825503355703, 629.194630872483, 855.7046979865769, 1249.9999999999995, 855.7046979865769, 629.194630872483, 792.2946295903203, 879.8556569730415, 1249.9999999999995, 874.548928
```

```
# Q-learning tests
# Set seed every time for consistent executions;
# comment out to get different random runs
random.seed(340)
print(Problem(deterministic_test).solve(ITERATIONS, True))
# print(Problem(deterministic_test).solve(10, True))
```

```
R D U R D
R R U U D
R U U D L
R U L L L
```

```
random.seed(340)
print(Problem(sampleMDP).solve(ITERATIONS, True))
```

```
D D U R D
R R U U D
U U U D D
U U L L L
```

```
random.seed(340)
print(Problem(very_slippy_test).solve(ITERATIONS, True))
```

```
D L U R D
R L U U L
D L U R D
```

```
    U R U R U
```

```
random.seed(340)
print(Problem(big_test).solve(ITERATIONS, True))
```

```
    L U R U L U R R U L
    U U U U U D U U U U
    U U U U U D U D U U
    U D U U U D D D L U
    R R U L L L L L U U
```

Once you're done, here are a few thought questions (19 points total):

3, 5 points) Suppose we are on the deterministic map where there is no sliding on ice, and performing value iteration until it converges. Supposing 0 < DISCOUNT_FACTOR < 1, how does the policy change if the discount factor changes to another value in that range (or does the policy change at all)? Why does that happen? What happens to the policy if DISCOUNT_FACTOR = 1?

TODO Changes to the discount factor would change the policy by increasing the value the policy places on rewards further away if the discount factor increases, or decreasing the value the policy places on rewards further away. If the discount factor is equal to 1, the policy would find the action that maximizes the total reward/utility for each space, regardless of how efficient it is. This could lead to the policy recommending actions that take much more moves to reach the gold.

4, 3 points) The value iteration MDP solver updates all squares an equal number of times. The Q-learner does not. Which squares might we expect the Q-learner to update the most?

TODO The squares we might expect the Q-learner to update the most would be the squares with gold. The reason we might expect the Q-learner to update the squares with gold the most is because the Q-learner emphasizes exploring/following paths with the highest reward that it has found. While every square could be explored by random choice, the Q-learner would learn about the positive value/utility of the gold spaces, causing the Q-learner to explore the paths leading to the gold spaces more than paths that lead to the pit spaces. Since each of the paths leading to the gold spaces ends with a gold space, the gold spaces would be updated the most by the Q-learner.

5, 11 points) Suppose we change the state information so that, instead of knowing its coordinates on the map, the agent instead knows just the locations of all rewards in a 5x5 square with the agent at the square center. Thus, at the start of every run, it may not know exactly where it is, but it knows what is in the vicinity. It also does not know the transition model.

a, 2 points) We can't use value iteration here. Why?

b, 4 points) How many state-action combinations are possible, assuming the contents of the agent's own square don't matter, and every other square could have a pit, gold, or an empty square as in the example maps? Is a lookup table of Q-values feasible if we allocate memory for each possible state-action combination? (Let's define "feasible" as "able to be stored in a gig or less of memory," assuming 64-bit values.)

c, 5 points) Let's suppose we want to instead generate Q-values with a classic neural network with a single hidden layer. The inputs are the contents of the 24 squares in the 5x5 square that the player is not in (we can encode gold = 1, nothing = 0, pit = -1). There are 10 hidden units. There are 4 output units corresponding to the 4 possible actions' Q-values. How much memory is required for the weights of this

network, assuming each is a 32-bit float (don't forget bias weights for each unit)? Comparing to part (b), is it more efficient in memory to use a lookup table for Q(s,a), or this neural network?

a) TODO We can't use value iteration here because we don't know the transitional model. Therefore, we can't apply the Bellman equation to accurately calculate the value for each space using value iteration. However, it would be possible to utilize a Q-learner model to explore this problem since it doesn't need to know the transitional model to explore the problem.

b) TODO The number of state-action combinations possible is the number of different states times the number of different actions. This would mean that the amount of memory required for the lookup table of Q-values would be $(5*5-1)*4=96$ values, each taking up 64 bits for a total of 6144 bits. This is because for each state (except for the agent's starting square), a Q learner would need to store the utility of every possible action. A lookup table of Q-values would be feasible if we allocate memory for each possible state-action combination.

c) TODO The amount of memory required for the weights of this classic neural network with a single hidden layer assuming each weight is a 32 bit float would be 9408 bits. There are $(24+1)*10+(10+1)*4=294$ weights in the neural network (each neuron has weights for each of its inputs + 1 for the bias weight). Since there are 294 weights, each taking up 32 bits, the total number of bits required to represent the weights of this classic neural network would be 9408. Comparing to part (b), the more efficient memory implementation would be to utilize a lookup table for Q(s, a).

Remember to submit your code on Blackboard as both an .ipynb (File->Download->.ipynb) and a PDF (Print->Save as PDF).