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:

In [1]:
```
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.

In [11]:
```python
""" "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 locat

    ...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"

    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 differer
            use_q (bool):  False means use MDP value iteration, true means u
        Returns:
            A Policy, in either case (what to do in each square; see class b
        """

        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 c
    action to take in each non-reward square (see MOVES constant at top of f
    """

    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, m
        return '\n{}\n'.format('\n'.join([' '.join(row) for row in self.best

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 checkin

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

    Args:
        move_probs (List[float]):  Transition probabilities for the ice (fro
        row, col (int, int):  location of agent before moving
        move (string):  The direction of move as a MOVES character (not an i
        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
    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 t
    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
            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 mdp_solve(problem, iterations):
    """ Perform value iteration for the given number of iterations on the MD

    Here, the squares with rewards can be initialized to the reward values,
    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
    """
    # TODO calculate the policy
    rows = len(problem.map)
    cols = len(problem.map[0])
    utilities = [[0 for _ in range(cols)] for _ in range(rows)]
    # Initialize utilities based on problem map
    for r in range(rows):
        for c in range(cols):
            if problem.map[r][c] == 'G':
                utilities[r][c] = GOLD_REWARD
            elif problem.map[r][c] == 'P':
                utilities[r][c] = PIT_REWARD

    # Value iteration
    for _ in range(iterations):
        new_utilities = copy.deepcopy(utilities)
        for r in range(rows):
            for c in range(cols):
                if problem.map[r][c] in ['G', 'P']:
                    continue  # Skip gold and pit cells, their utility remai
                action_utilities = []
                for move, action in enumerate(MOVES):  # Check utility for e
                    total_expected_utility = 0
                    for displacement, prob in enumerate(problem.move_probs):
                        new_r, new_c = roll_steps([prob], r, c, action, rows
                        total_expected_utility += prob * utilities[new_r][ne
                    action_utilities.append(total_expected_utility)
                new_utilities[r][c] = max(action_utilities)  # Bellman updat

        utilities = new_utilities  # Update utilities after all states consi

    # Generate policy from utilities
    policy = Policy(problem)
    for r in range(rows):
        for c in range(cols):
            if problem.map[r][c] in ['-']:
                best_action = None
                best_utility = float('-inf')
                for move, action in enumerate(MOVES):
                    expected_utility = 0
                    for displacement, prob in enumerate(problem.move_probs):
                        new_r, new_c = roll_steps([prob], r, c, action, rows
                        expected_utility += prob * utilities[new_r][new_c]
                    if expected_utility > best_utility:
                        best_action = action
                        best_utility = expected_utility
                policy.best_actions[r][c] = best_action
    return policy

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, f
    the agent until a reward is reached or MAX_MOVES moves have been made.
    is sitting on a reward, update the utility of each move from the space t
    and end the iteration.  (For simplicity, the agent also does this if jus
    The agent does not "know" reward locations in its Q-values before encour
    and "discovering" the reward.

    Note that texts differ on when to pay attention to this reward - this co
    convention of scoring rewards of the space you are moving *from*, plus c
    of where you landed.
    Assume epsilon-greedy exploration.  Leave reward letters as-is in the po
    to make it more readable.

    Args:
        problem (Problem):  The environment
        iterations (int):  The number of runs from random start to reward en
    Returns:
        A Policy for the map
    """
    # TODO
    rows, cols = len(problem.map), len(problem.map[0])
    # Initialize Q-table: rows x cols x ACTIONS (4 for up, right, down, left
    Q = [[[0.0 for _ in range(ACTIONS)] for _ in range(cols)] for _ in range

    for _ in range(iterations):
        # Random starting point
        r, c = random.choice([(r, c) for r in range(rows) for c in range(col
        for move_count in range(MAX_MOVES):
            # Decide whether to explore or exploit
            if random.random() < EXPLORE_PROB:
                action = random.randint(0, ACTIONS - 1)  # Explore: random a
            else:
                action = Q[r][c].index(max(Q[r][c]))  # Exploit: choose best

            # Move and calculate new state
            new_r, new_c = roll_steps(problem.move_probs, r, c, MOVES[action

            # Determine the reward
            if problem.map[new_r][new_c] == 'G':
                reward = GOLD_REWARD
            elif problem.map[new_r][new_c] == 'P':
                reward = PIT_REWARD
            else:
                reward = 0  # Default reward for other moves

            # Update Q-value for the action taken
            if problem.map[new_r][new_c] in ['G', 'P']:  # Terminal state
                # Set all Q-values for the terminal state to its correspondi
                Q[new_r][new_c] = [reward] * ACTIONS
                # End this trial since the agent has reached a terminal stat
                break
            else:
                # Standard Q-learning update rule
                best_future_q = max(Q[new_r][new_c])
                Q[r][c][action] += LEARNING_RATE * (reward + DISCOUNT_FACTOR

            # Update state to new state
            r, c = new_r, new_c

    # Generate policy from Q-table
    policy = Policy(problem)
    for r in range(rows):
        for c in range(cols):
            if problem.map[r][c] == '-':
                best_action_index = Q[r][c].index(max(Q[r][c]))
                policy.best_actions[r][c] = MOVES[best_action_index]
            else:
                policy.best_actions[r][c] = problem.map[r][c]  # Leave rewar

    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-learn

    Should use the LEARNING_RATE and DISCOUNT_FACTOR.

    Args:
        rewards (List[List[float]]):  Reward amounts built into the problem
        utilities (List[List[List[float]]]):  The Q-values for each action f
                                              (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 co
    Returns:
        float - the new Q-value for the space we moved from
    """
    # TODO
        # Extract the current Q value for the state-action pair
    current_q_value = utilities[r][c][movenum]
    # Calculate the reward for moving to the new state
    reward = rewards[new_r][new_c]
    # Find the maximum Q-value among all possible actions from the new state
    max_future_q = max(utilities[new_r][new_c])
    # Compute the updated Q-value using the Q-learning formula
    updated_q_value = current_q_value + LEARNING_RATE * (reward + DISCOUNT_F


    return updated_q_value
```

In [12]:
```
deterministic_test = """1.0
- - P - -
- - G P -
- - P - -
- - - - -"""
```

In [13]:
```
# 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 - -
- - - - -"""
```

In [14]:
```
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"""
```

In [15]:
```
# MDP value iteration tests
print(Problem(deterministic_test).solve(ITERATIONS, False))
```
```
U U P U U
U U G P U
U U P R U
U U R U U
```

In [16]:
```
print(Problem(sampleMDP).solve(ITERATIONS, False))
```
```
U U P U U
U U G P U
U U P R U
U U R U U
```

In [17]:
```
print(Problem(very_slippy_test).solve(ITERATIONS, False))
```
```
U U P U U
U U G P U
U U P R U
U U R U U
```

In [18]:
```
print(Problem(big_test).solve(ITERATIONS, False))
```
```
U P U G U P U U G U
P G U P U R U P U U
P P U P P U P D P U
P R U P P U R U L P
R U U R R U U U P G
```

In [19]:
```
# 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))
```
```
U U P U U
U U G P U
U U P U U
U U U U U
```

In [20]:
```
random.seed(340)
print(Problem(sampleMDP).solve(ITERATIONS, True))
```
```
U U P U U
U U G P U
U U P U U
U U U U U
```

In [21]:
```
random.seed(340)
print(Problem(very_slippy_test).solve(ITERATIONS, True))
```

```
U U P U U
U U G P U
U U P U U
U U U U U
```

In [22]:
```
random.seed(340)
print(Problem(big_test).solve(ITERATIONS, True))
```

```
U P U G U P U U G U
P G U P U U U P U U
P P U P P U P U P U
P U U P P U U U U P
U U U U U U U U P G
```

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?

Answer:

When performing value iteration on a deterministic map:

If 0 < DISCOUNT_FACTOR < 1: Changing the discount factor alters the balance between valuing immediate versus future rewards. A lower discount factor makes the policy favor immediate rewards more, becoming short-sighted. A higher discount factor makes the policy value future rewards more, becoming far-sighted. Therefore, the policy can change depending on whether immediate or future rewards are prioritized.

If DISCOUNT_FACTOR = 1: The policy treats all rewards, immediate and future, equally. This can lead to a focus on maximizing long-term rewards without regard for immediacy, which might not always be practical. However, this setting can cause convergence issues and might not accurately reflect realistic decision-making scenarios where future uncertainties should diminish the value of distant rewards.

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?

Answer:

The Q-learner is likely to update squares that are frequently visited more often. These often include starting points, common paths to rewards, and areas near the initial positions due to random exploration. Squares near rewards or common routes may also see more updates due to their higher attractiveness for learning optimal paths.

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)

Value iteration requires knowledge of the exact state (coordinates on the map) and the transition model to update the utilities of each state based on the possible outcomes of actions. If the agent only knows about rewards within a 5x5 vicinity and lacks the exact coordinates or transition model, it can't predict the outcome of its actions accurately for the whole map, making value iteration inapplicable.

b)

There are 3 possible conditions for each of the 24 squares around the agent (pit, gold, empty), so the total number of state configurations is 3^{24}. With 4 possible actions, there are $3^{24} * 4$ state-action combinations. Assuming each Q-value is a 64-bit value (8 bytes):

Total memory = $3^{24} * 4 * 8$ bytes.

We need to calculate this to see if it's under one gigabyte (which is 2^{30} bytes.)

The total number of state-action combinations requires approximately 8417.06 GB of memory, which is not feasible as it exceeds our 1 GB limit.

c)

For the neural network:

Input to hidden layer weights: $24 * 10 = 240$ weights.

Bias weights for hidden layer: 10.

Hidden to output layer weights: $10 * 4 = 40$ weights.

Bias weights for output layer: 4.

Total weights = 240 + 10 + 40 + 4 = 294.

Since each weight is a 32-bit float (4 bytes):

Total memory for NN = $294 * 4$ bytes.

The memory required for the weights of the neural network is approximately 1.15 KB, which is significantly less than the memory needed for the lookup table of Q-values.

Comparing the two, it is far more efficient in terms of memory to use a neural network rather than a lookup table for storing Q-values.

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