Ian Wilhite

AERO 489: Selva, Valasek

3/4/2025

Question 1: Formulate this problem as a Markov Decision Process. Specifically, provide the states, actions, rewards, and transition model.

State:

- The location of the rover within the world (x,y)
- The danger of the cells, as stored within the grid.

Actions:

- Move Up, (x,y) -> (x, y + 1)
- Move Right, (x,y) -> (x + 1, y)
- Move Down, (x,y) -> (x, y - 1)
- Move Left, (x,y) -> (x - 1, y)

Rewards:

- Move cost: -0.05 (included on unsuccessful moves)
- Goal reward: +1.00
- Crater cost: -1.00

Transition Model (for all actions):

- Move successfully -> 80%
- Stay in current cell -> 10%
- Slide towards center of grid -> 10%

Question 2: Write code in Python that implements the environment. This should include:

- A function to create a fixed test world (e.g., start cell in (1,1), goal in (5,5), danger cells in (1,4), (2,4), (2,2), (3,4), crater in (4,3)).

```python
def create_test_problem_5x5(transition): # generates test problem instance
    grid = np.empty((5, 5), dtype=object)
    for i in range(5):
        for j in range(5):
            science = 0
            danger = 0
            grid[i, j] = GridCell(science=science, danger=danger)
    grid[0][3].danger = 0.5
    grid[1][3].danger = 0.8
    grid[1][1].danger = 0.9
    grid[2][3].danger = 0.7
    grid[3][2].danger = 0.5
    grid[4][4].science = 1.0
    initial_state = RoverState(0, 0)
    return Problem(initial_state, goal_test, grid, transition_function=transition)
```

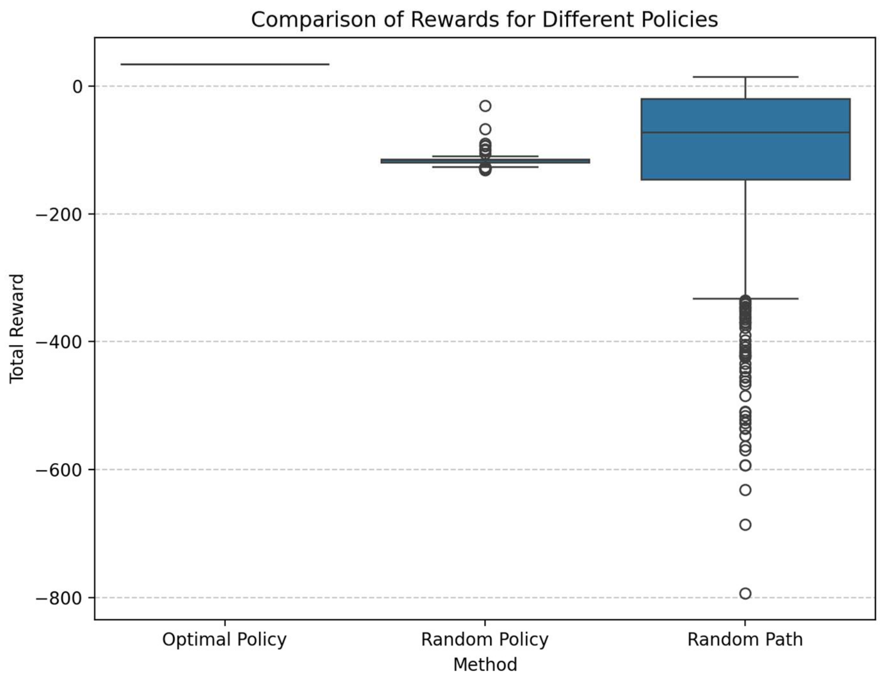
- A function to create a random world of this type

```python
def generate_random_problem(transition): # generates random problem instance of size N
    n = N
    grid = np.empty((n, n), dtype=GridCell)
    n_unsafe = 0

    # Step 1: Initialize the grid with danger values
    for i in range(n):
        for j in range(n):
            if random.random() < percent_cells_with_danger:  # 20% chance of an unsafe cell
                danger = random.uniform(0.50001, 1)
                n_unsafe += 1
            else:  # Safe cell
                danger = random.uniform(0, 0.49999)

            grid[i, j] = GridCell(science=0, danger=danger)

    # Step 2: Set the goal cell within the safe cells (with danger < 0.5)
    (goal_x, goal_y) = np.floor(random.uniform(0, n-1)), np.floor(random.uniform(0, n-1))
    while (True): # bogo find, but it's fine
        (start_x, start_y) = np.floor(random.uniform(0, n-1)), np.floor(random.uniform(0, n-1))
        if goal_x != start_x and goal_y != start_y:
            grid[goal_x, goal_y].science = 1.0
            grid[goal_x, goal_y].danger = 0.0
            break

    # Step 3: Find a random initial position that corresponds with a safe cell
    while True: # bogo find, but it's fine
        initial_x, initial_y = random.randint(0, n-1), random.randint(0, n-1)
        if grid[initial_x, initial_y].danger <= 0.5:  # Only pick a safe cell
            initial_state = RoverState(initial_x, initial_y)
            break

    return Problem(initial_state, goal_test, grid, transition)
```

- Functions implementing the transition model and the reward function. Given a current state and action, they should return the next state and the reward respectively (or you can do this in the same function).

```python
def transition_model(state, action, grid):
    """ Implements the stochastic movement model with 80% success, 10% stay, and 10% slide. """

    x, y = state.x, state.y
    # Define sliding move (towards the center of the grid)
    center_x, center_y = (N - 1) / 2, (N - 1) / 2 # 2, 2 for 5x5 grid

    # Apply probability
    rand_value = random.random()
    if rand_value < 0.8:  # 80% chance to move as intended
        new_x = x + action[0]
        new_y = y + action[1]
    elif rand_value < 0.9:  # 10% chance to stay in place
        new_x, new_y = x, y
    else:  # 10% chance to slide towards the center
        if x < center_x: new_x = x + 1
        elif x > center_x: new_x = x - 1
        if y < center_y: new_y = y + 1
        elif y > center_y: new_y = y - 1

    # Check if move is valid (not out of bounds)
    if 0 <= new_x < 5 and 0 <= new_y < 5:
        return RoverState(new_x, new_y)
    return state  # Stay in place if movement is invalid

def reward_function(state, grid):
    """ Returns the reward associated with the given state. """
    if grid[state.x][state.y].is_goal:
        return 1.00  # Goal reward
    if grid[state.x][state.y].danger >= 0.5:
        return -1.00  # Crater penalty
    return -0.05  # Move cost
```

Question 3: Write code for an agent that follows a given policy, specified as a look up table (state, action). Run an agent using a random policy (i.e., that moves randomly in this environment until reaching a terminal state) in the test problem 1000 times. Make a boxplot of the total undiscounted rewards collected by the agent.

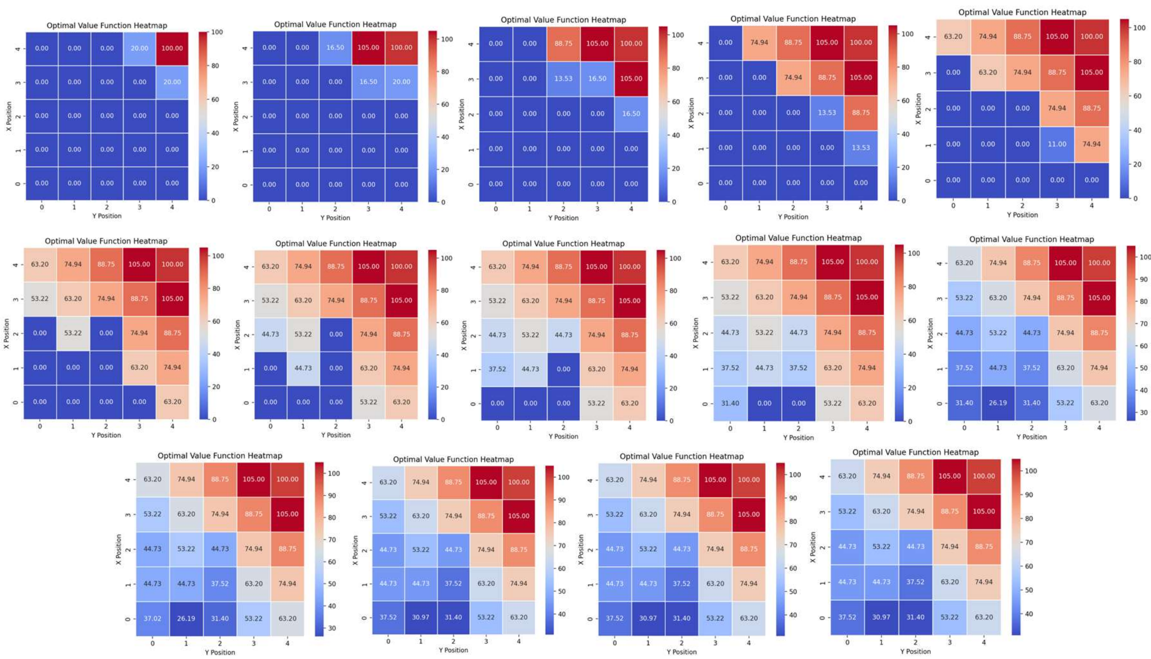


Random policy method represents that a policy was randomly generated then applied, if an infinite loop occurred then a penalty of -100 was applied and the episode terminated. The high cost in the random policy was often a result of the repeated high failure penalty. The random policy method has a cluster just below -100, because its failure is often due to the -100 failure penalty due to looping.

Random path indicates that a random move was applied at each possible situation, where the agent was allowed to cross back over the same location multiple times. The high cost in the random path was often due to the accumulation of the movement cost for 100+ moves to solve the problem. The random path is dispersed much more broadly as it has the chance to converge sooner, and a much lower spread as it allows for repeat cell visits.

The optimal policy agent only has a single value because there is no random element to the design process.

Question 4: Write an algorithm that implements value iteration. Compute the optimal value function and the optimal policy for the test problem using your algorithm (show the outputs visually). Evaluate an agent following the optimal policy and compare the total rewards obtained with those of the agent following a random policy



The optimal policy unsurprisingly performs much better than the other two random-based models at solving the problem and avoiding error. It is worth noting that there is an error in the policy generated in the top middle where the policy directs the agent to move left, when it should move right. This error might be caused by a discrepancy between the values being displayed and the combination of (reward + discount_factor * value[next_state]) which could be caused by the presence of a crater. This type of local maximum would result in an infinite loop in the policy as it would converge on a non-goal value.

```
[Running] python -u "c:\Users\ianwi\OneDrive\Documents\S4\AERO489\hw3.py"
Starting Random Policy Simulation...
Optimal Policy:
> > < > G
^ ^ > > ^
> > v > ^
v > > > ^
> > > > ^
Average Reward (Optimal Policy): 34.95
Average Reward (Random Policy): -116.26707
Average Reward (Random Path): -104.8485

[Done] exited with code=0 in 5.632 seconds
```