CSCI 264

Markov Decision Process

Project 1

Janaki Panneerselvam

Spring 2023

Introduction:

In this study, we explore a Markov Decision Process (MDP) where an agent navigates a grid-like environment of size n x n. The agent's goal is to reach a certain position with a high reward while avoiding walls, which incur significant penalties. The agent has two modes of movement, each with its inherent risks and rewards: a short, safer distance (d=1) and a longer, riskier one (d=2). For both distances, moving in a given direction doesn't guarantee arrival at the intended spot due to possible sliding. Sliding can result in the agent veering off course or even staying in place. The risk of sliding increases with the longer distance but also comes with a higher reward. This problem presents an interesting exploration of risk-reward dynamics in decision-making processes.

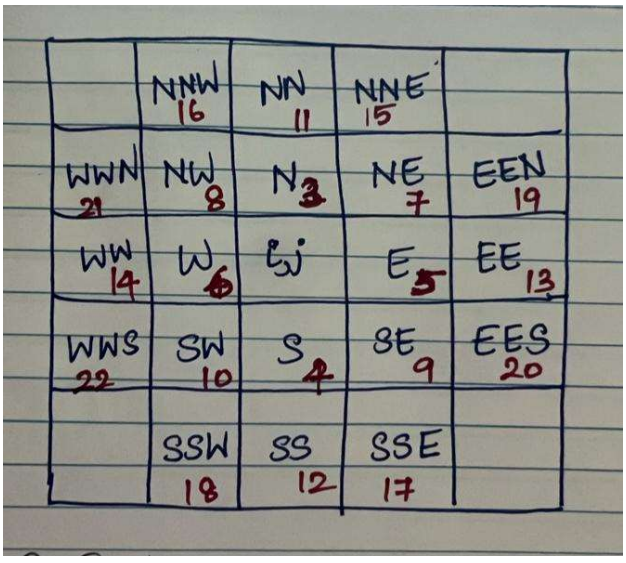
Implementation:

In our solution, we implemented several methods to solve the given Markov Decision Process (MDP) problem.

1. **Matrix Reading and Preparation:** This method reads the maze data from CSV files and prepares the matrix representation of the maze, including the locations of the walls, start and end points.

2. **Action Matrix Creation:** Here, we defined an action matrix that comprises all possible actions the agent could take. These include movement in four directions - up, down, left, and right, each with two possible distances (d=1, d=2). Action matrix has the following actions for each cell of the maze.

Functions implemented: `allowed_state_actions & actions_with_walls`



3. **Probability Transition Matrix Computation:** We implemented a method to compute the transition probabilities for each state-action pair considering the actions defined in the action matrix. This involved accounting for the potential of the agent sliding to unintended states.

```
move_north
move_south
move_east
move_west
```

4. **Value Iteration Algorithm:** A key component of our solution was the implementation of the Value Iteration algorithm. This algorithm iteratively updates the estimated value of each state until the value function converges to the optimal one.

The main program will find the max and update the matrix.

5. **Policy Extraction:** Once the optimal value function is computed, we extract the optimal policy that the agent should follow to maximize its reward. Optimal policy extracted after value iteration is converged. The optimal policy is again under the probability slip of 0.2 percentage. So, the movement may not end in the intended space.

6. **Path Visualization**: Lastly, we implemented a method for visualizing the agent's path through the maze. This involved plotting the maze and the agent's path and adding graphical elements such as diamonds to indicate the agent's position at different points in time.

```
display_maze will display all the maze with optimal policy.
```

This combination of methods provided a comprehensive solution to the MDP problem, allowing us to find an optimal policy for the agent to navigate the maze while considering the inherent risks and rewards of its actions.

Results:

Maze 1:

```
prob_d1 = [0.1,0.7,0.1,0.1]
prob_d2 = [0.1,0.5,0.1,0.1,0.2]
```
Start = (1,1)
End = (5,5)

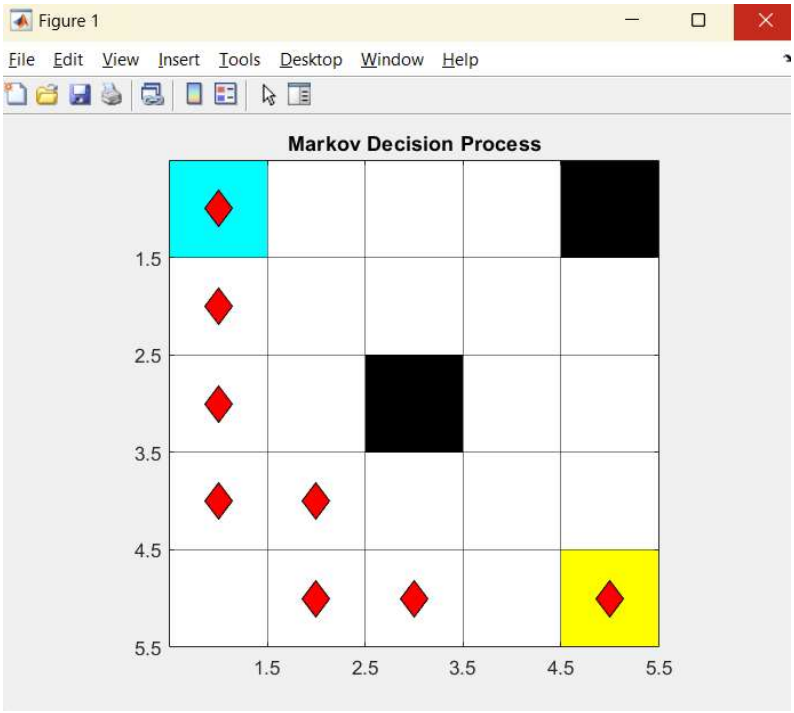
Discussion of the results:

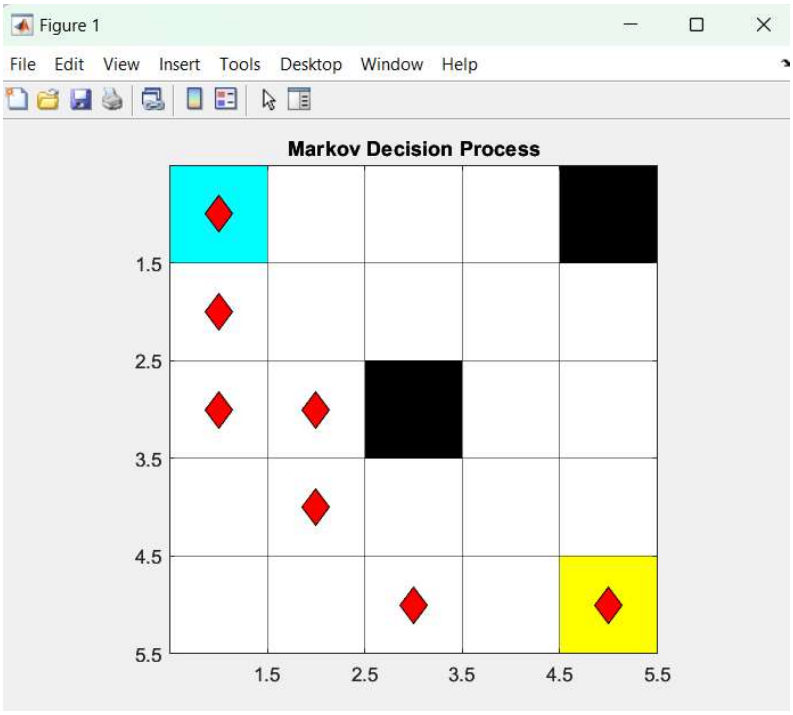
The selection of the optimal policy for the path is driven by the Value Iteration matrix, considering both d=1 and d=2. The main goal is to successfully reach the terminal state that holds a reward of 100. The following matrices illustrate the outcome of Value Iteration for d=1 and d=2.

The process to determine the Optimal Path involves the following steps:

Starting from position (1,1), the next optimal position is identified by selecting the maximum value among all permissible value iterations from the current position, (1,1). In the illustration below, we have marked the subsequent optimal probability in green for clarity.

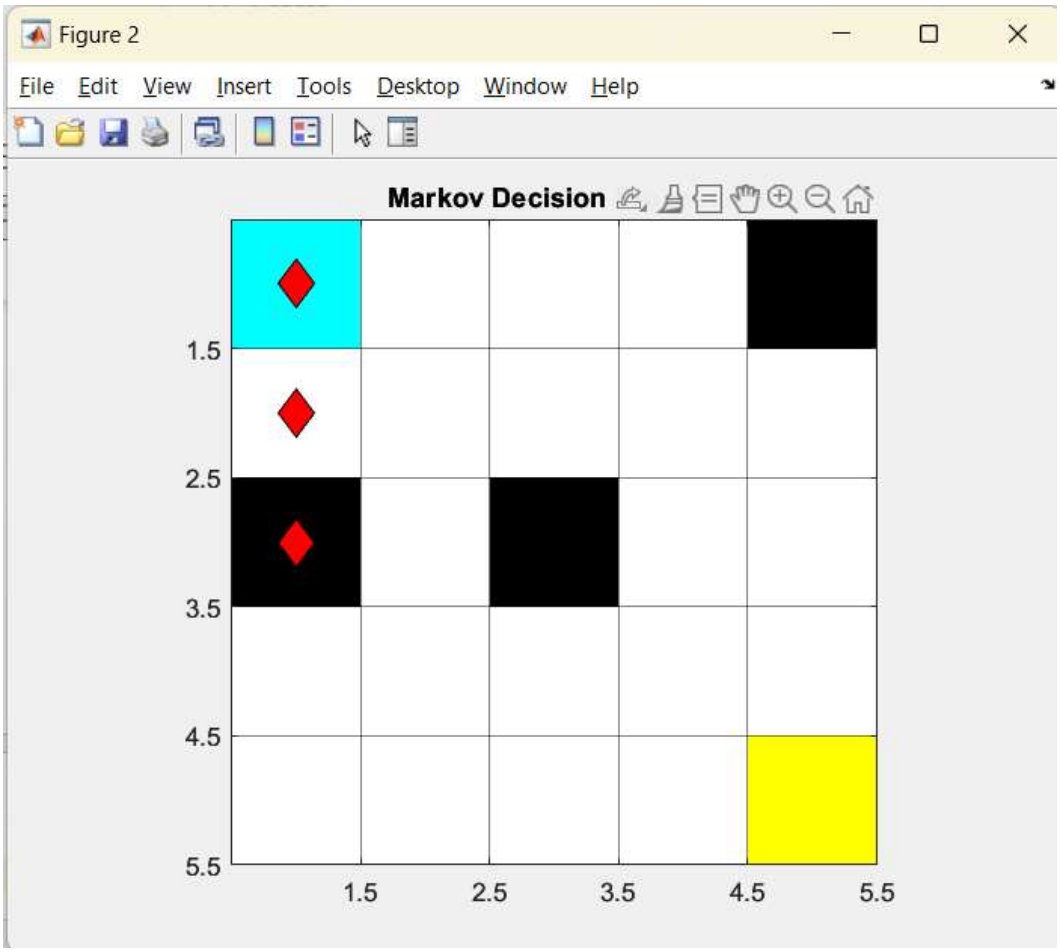
Below is the result of two different run of the MDP algorithm.





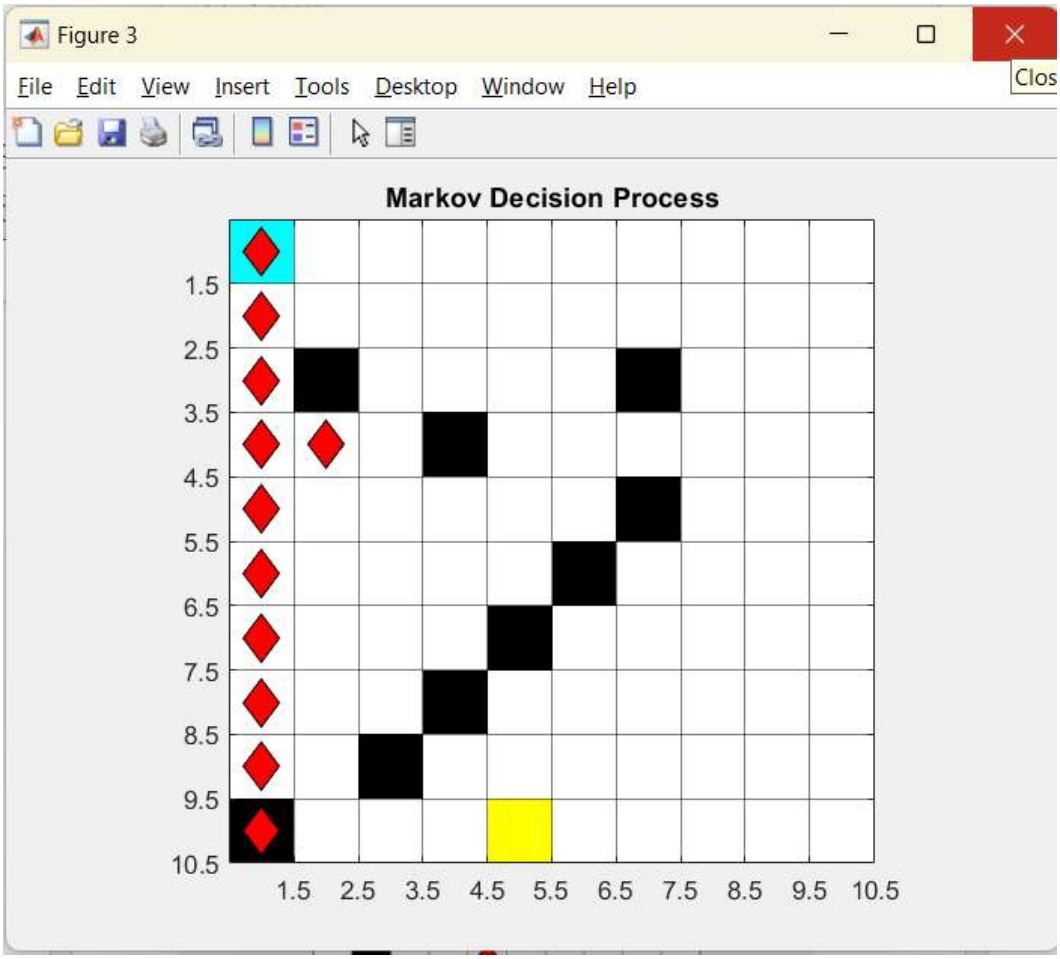
Maze 2

```
prob_d1 = [0.1,0.7,0.1,0.1]
prob_d2 = [0.1,0.5,0.1,0.1,0.2]
```



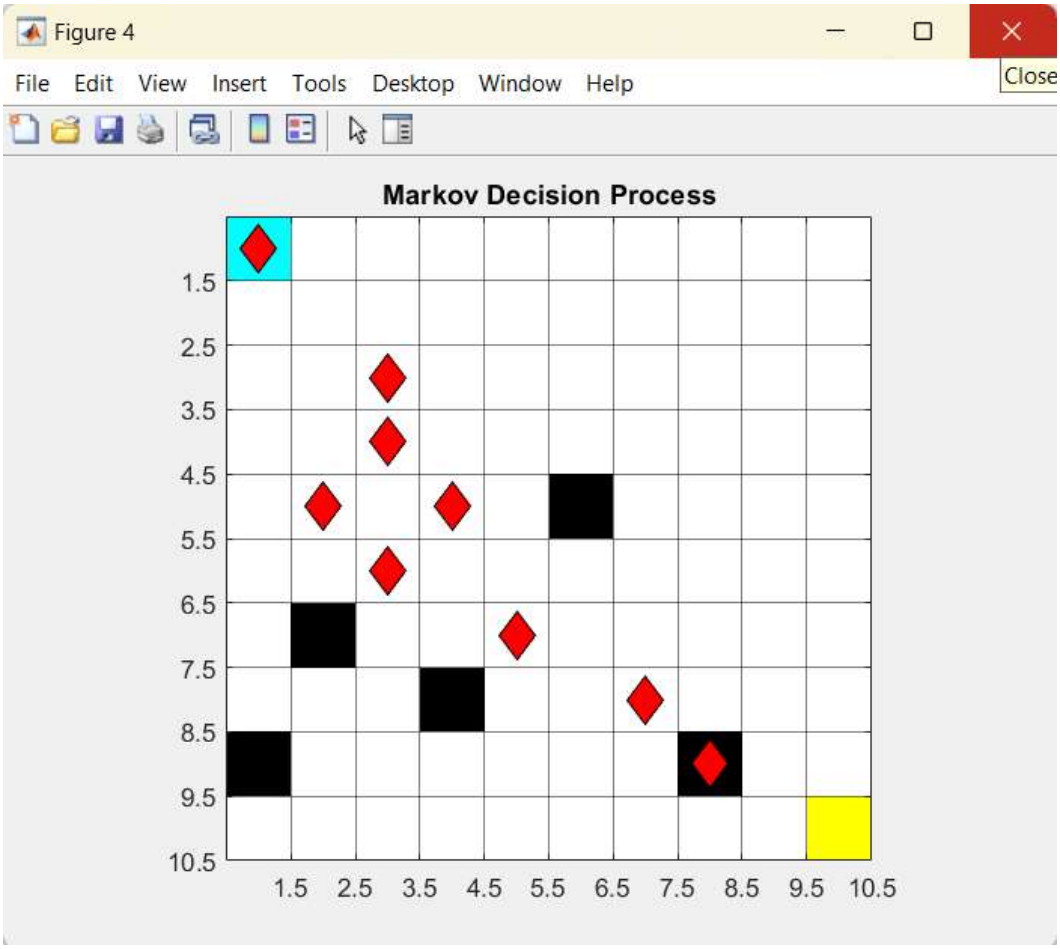
Maze 3:

```
prob_d1 = [0.1,0.7,0.1,0.1]
prob_d2 = [0.1,0.5,0.1,0.1,0.2]
```



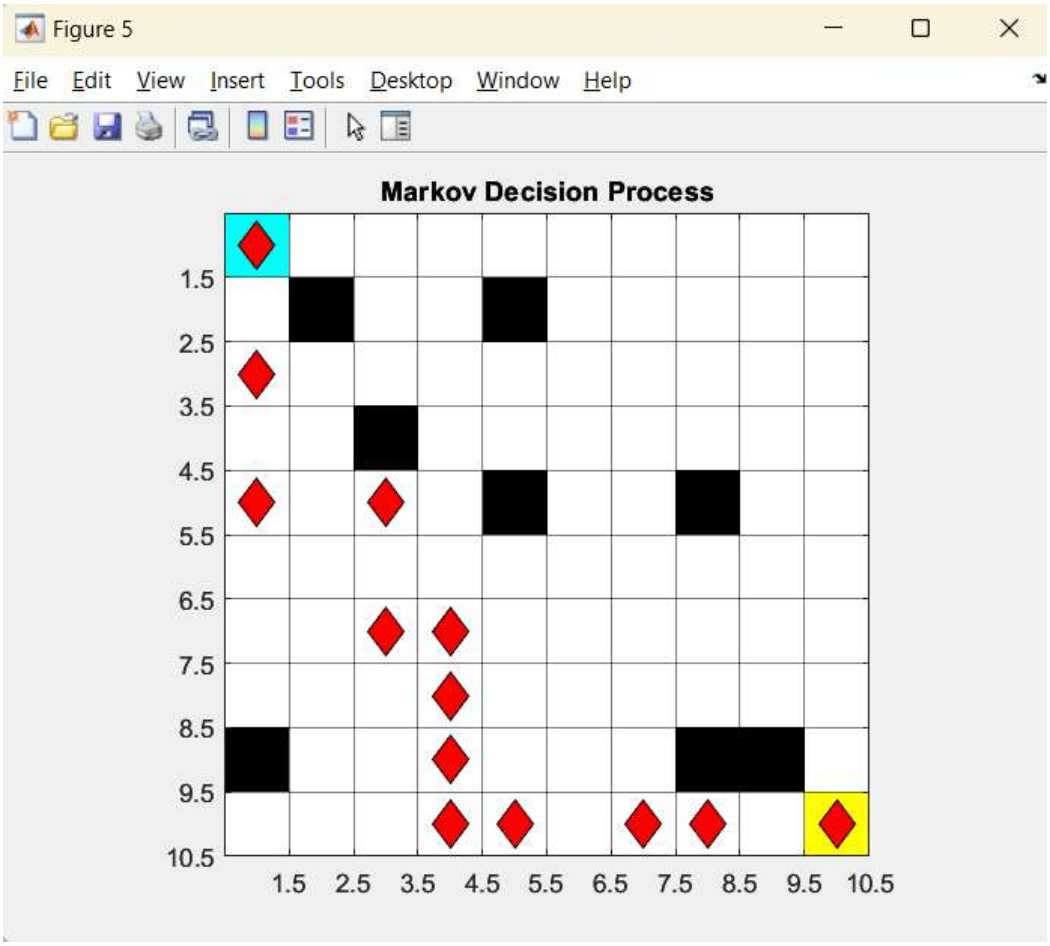
Maze 4:

```
prob_d1 = [0.1,0.7,0.1,0.1]
prob_d2 = [0.1,0.1,0.7,0.05,0.05]
```



Maze 5:

```
prob_d1 = [0.1,0.7,0.1,0.1]
prob_d2 = [0.1,0.1,0.7,0.05,0.05]
```



Maze 6:

```
prob_d1 = [0.1,0.7,0.1,0.1]
prob_d2 = [0.1,0.1,0.7,0.05,0.05]
```