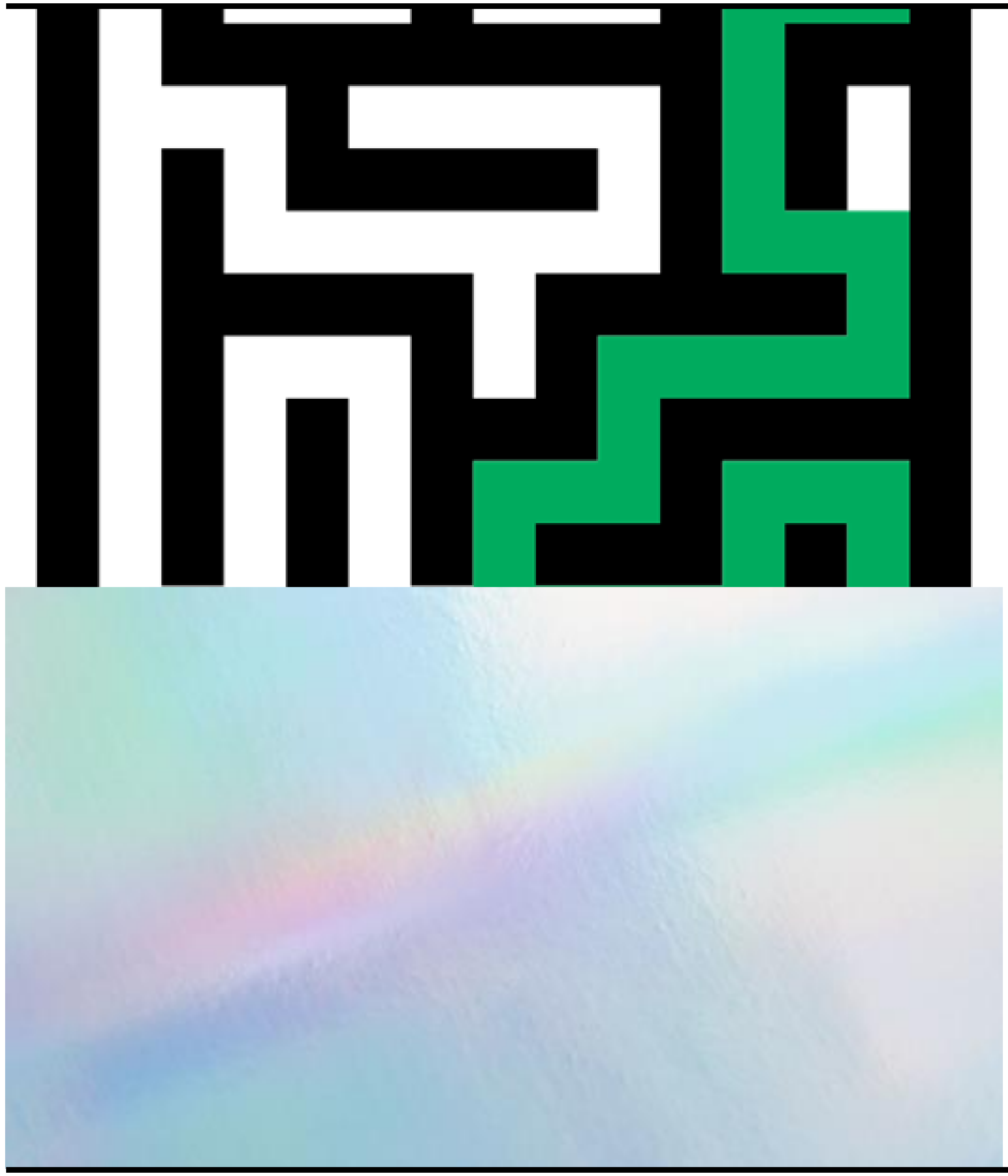


Stochastic Shortest path MAZE

Sreevidya C

CB.EN.D*ECE21012-FT

Introduction

Stochastic shortest path (SSP) problems involve finding the shortest path between two nodes in a graph,

The length of each edge is a random variable.

To implement SSP in Python,

NetworkX and perform Monte Carlo simulation to estimate the expected value of the path length.

Create

Create a graph using NetworkX:

Specify

Specify edge weights:

Perform

Perform Monte Carlo simulation

Compute

Compute expected value:

Monte Carlo Simulation

Stochastic Shortest Path (SSP) implementation is where the expected value of the path length is estimated through simulation. The idea behind Monte Carlo simulation is to generate a large number of random samples of the weights of the edges in the graph and use these samples to approximate the expected value of the path length.

Initialize the simulation: In this step, you set the number of simulations you want to run. For example, you might want to run 1000 simulations.

Sample edge weights: For each simulation, generate random samples of the edge weights. The edge weights represent the random variable associated with the length of each edges

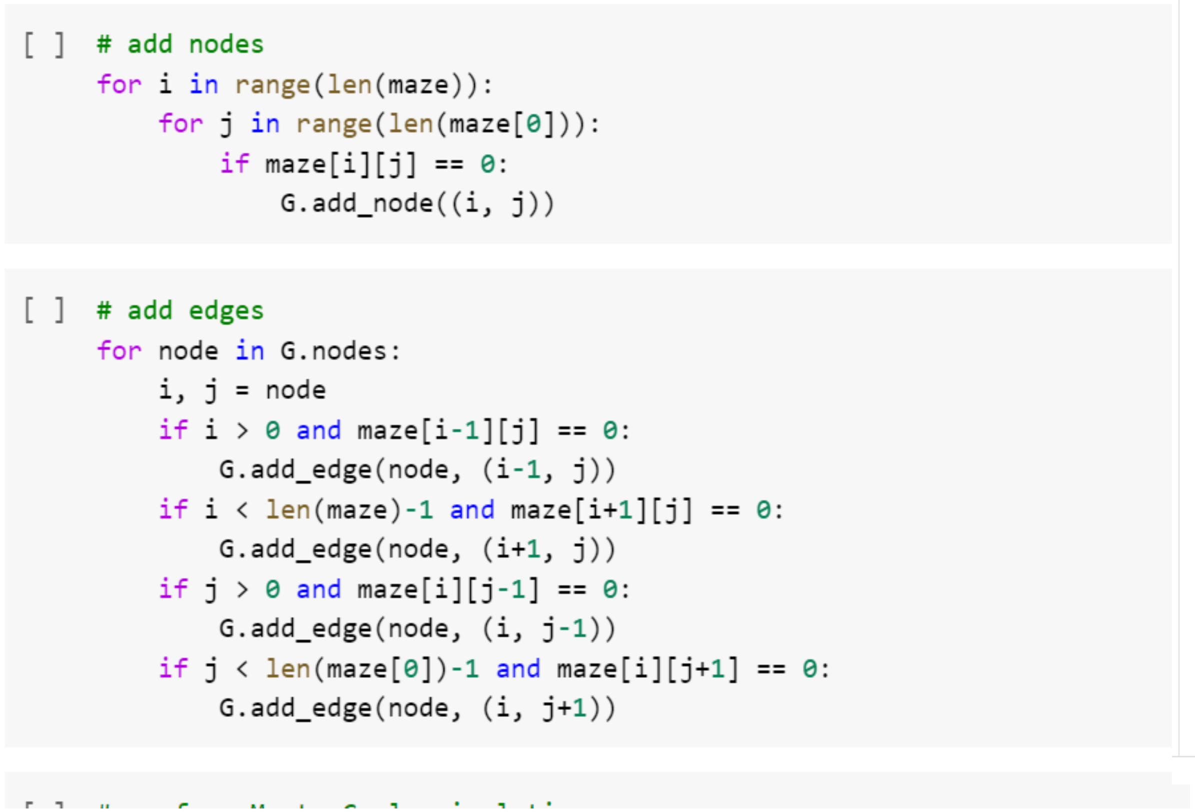
Find the shortest path: In each simulation, shortest path algorithm such as Dijkstra's algorithm to find the shortest path between the two nodes of interest. The edge weights used in this step are the ones generated in the previous step.

Record the path length: After finding the shortest path, record its length. This length represents the random variable associ ated with the path length in this simulation.

Repeat the process: Repeat the above steps (sampling edge weights, finding the shortest path, and recording the path length) for each of the simulations.

Compute the expected value: After all simulations are completed, you compute the expected value of the path length by averaging the lengths of all the shortest paths found in the simulations. This expected value represents an estimate of the average length of the shortest path in the graph.

Iterate over all nodes in the graph and add edges between them if they are adjacent in the maze and their corresponding cells are accessible

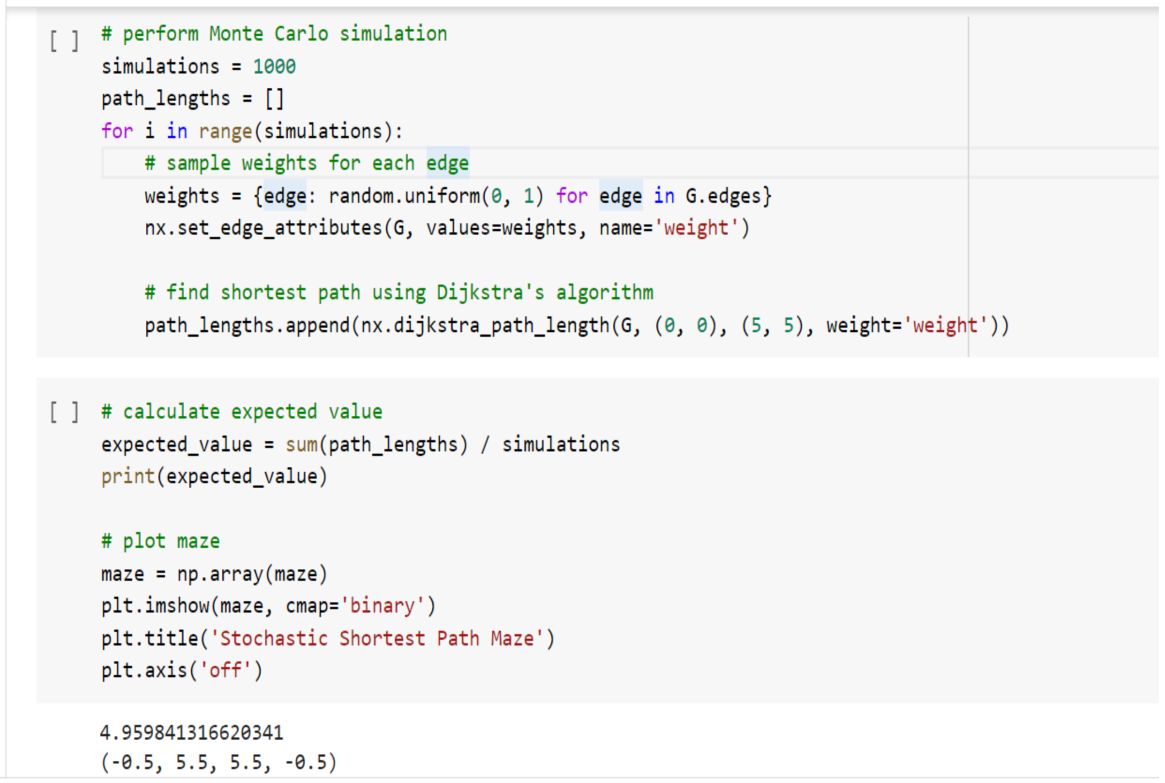


```
[ ] # add nodes
    for i in range(len(maze)):
        for j in range(len(maze[0])):
            if maze[i][j] == 0:
                G.add_node((i, j))
```

```
[ ] # add edges
    for node in G.nodes:
        i, j = node
        if i > 0 and maze[i-1][j] == 0:
            G.add_edge(node, (i-1, j))
        if i < len(maze)-1 and maze[i+1][j] == 0:
            G.add_edge(node, (i+1, j))
        if j > 0 and maze[i][j-1] == 0:
            G.add_edge(node, (i, j-1))
        if j < len(maze[0])-1 and maze[i][j+1] == 0:
            G.add_edge(node, (i, j+1))
```

```
[ ] # perform Monte Carlo simulati
```

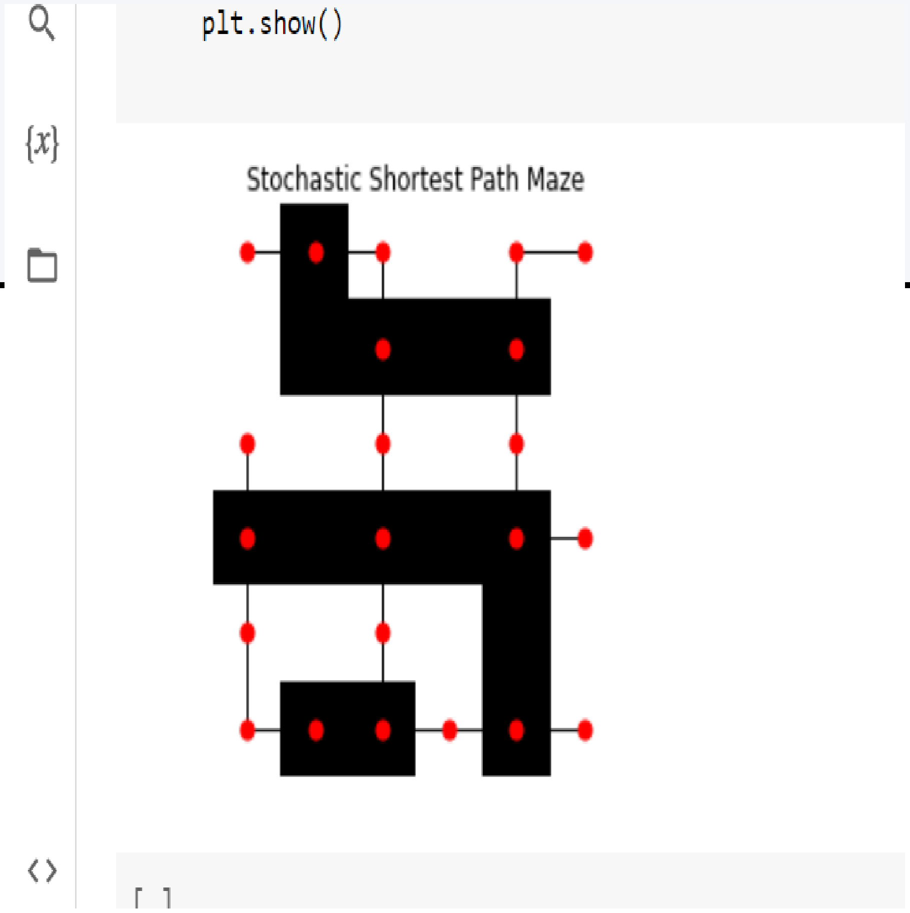
Monte carlo Simulation



```
[ ] # perform Monte Carlo simulation
    simulations = 1000
    path_lengths = []
    for i in range(simulations):
        # sample weights for each edge
        weights = {edge: random.uniform(0, 1) for edge in G.edges}
        nx.set_edge_attributes(G, values=weights, name='weight')

        # find shortest path using Dijkstra's algorithm
        path_lengths.append(nx.dijkstra_path_length(G, (0, 0), (5, 5), weight='weight'))
```

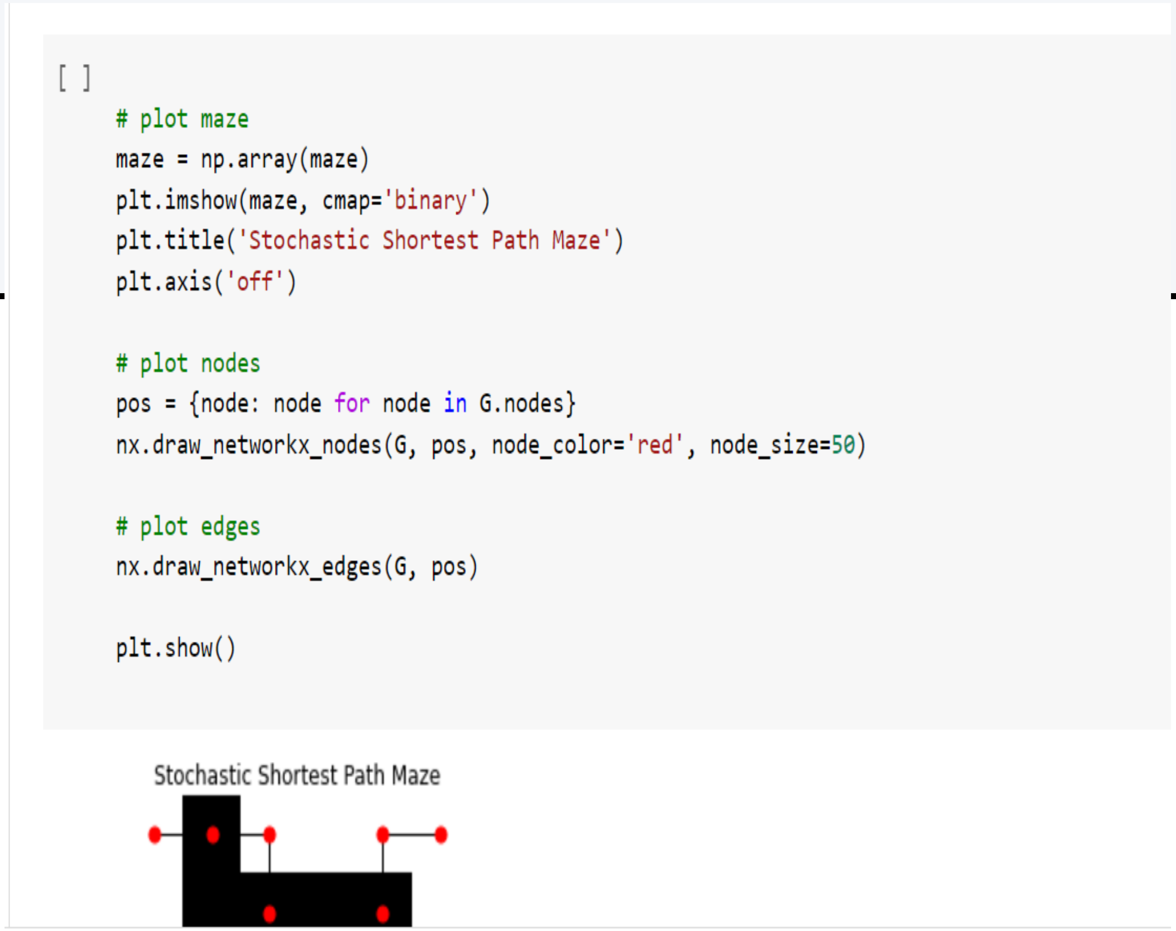
```
[ ] # calculate expected value
    expected_value = sum(path_lengths) / simulations
    print(expected_value)

    # plot maze
    maze = np.array(maze)
    plt.imshow(maze, cmap='binary')
    plt.title('Stochastic Shortest Path Maze')
    plt.axis('off')
```

```
    4.959841316620341
    (-0.5, 5.5, 5.5, -0.5)
```

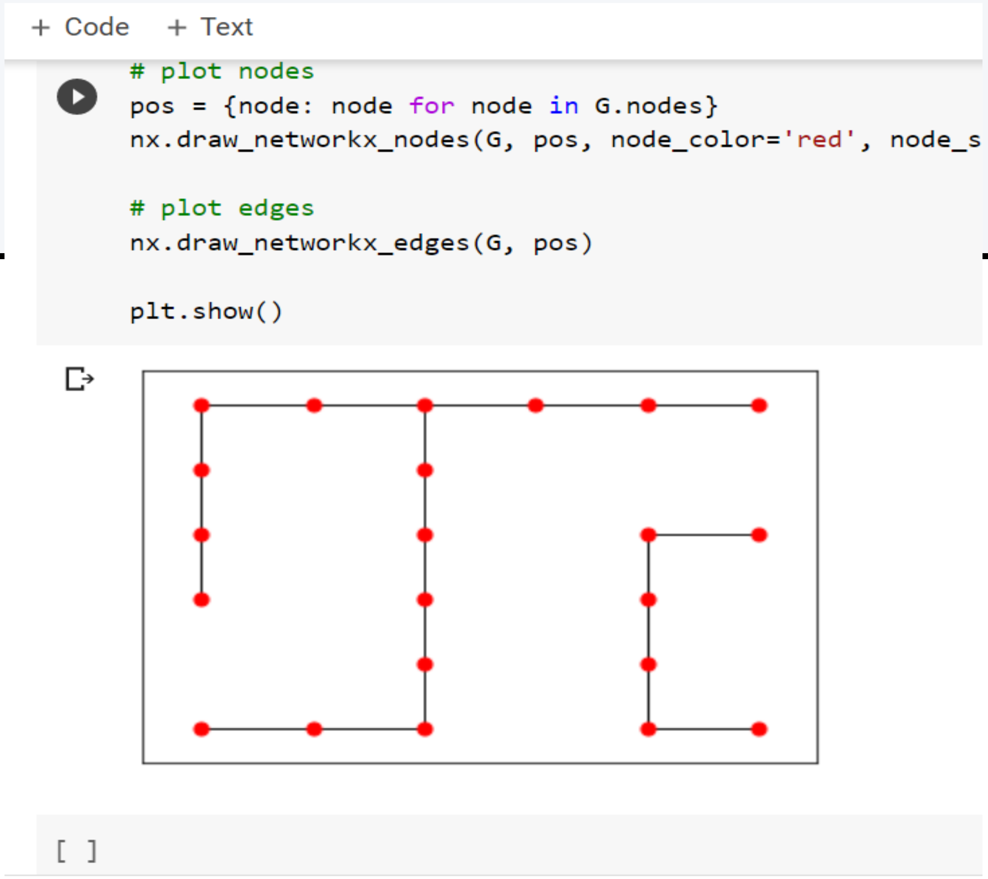


```
plt.show()
```

```
[ ]
```

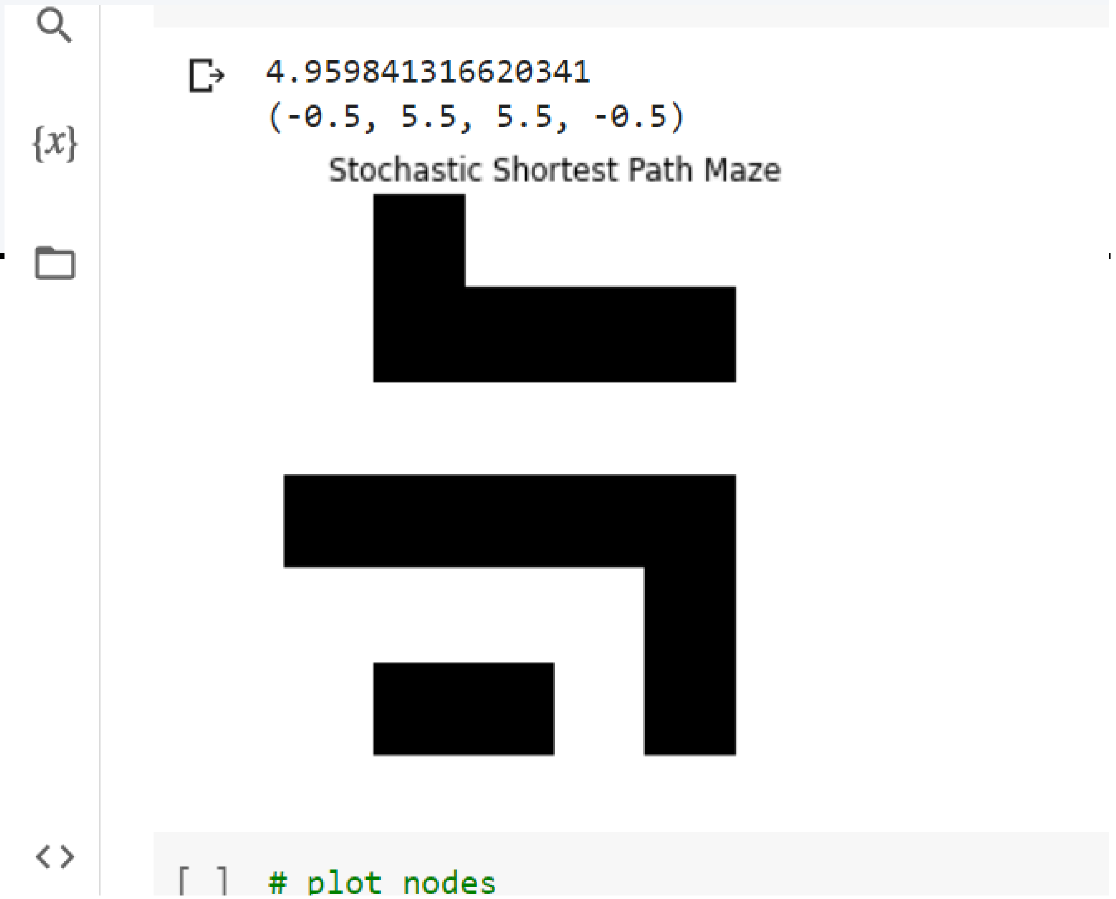


```
[ ]
    # plot maze
    maze = np.array(maze)
    plt.imshow(maze, cmap='binary')
    plt.title('Stochastic Shortest Path Maze')
    plt.axis('off')

    # plot nodes
    pos = {node: node for node in G.nodes}
    nx.draw_networkx_nodes(G, pos, node_color='red', node_size=50)

    # plot edges
    nx.draw_networkx_edges(G, pos)

    plt.show()
```



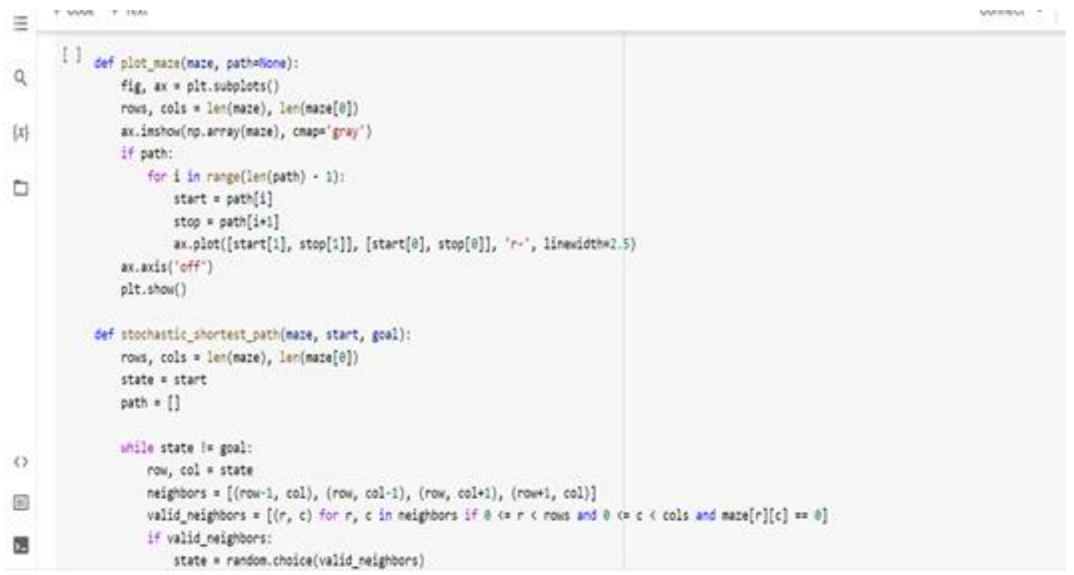
```
# plot nodes
pos = {node: node for node in G.nodes}
nx.draw_networkx_nodes(G, pos, node_color='red', node_s

# plot edges
nx.draw_networkx_edges(G, pos)

plt.show()
```

```
[ ]
```

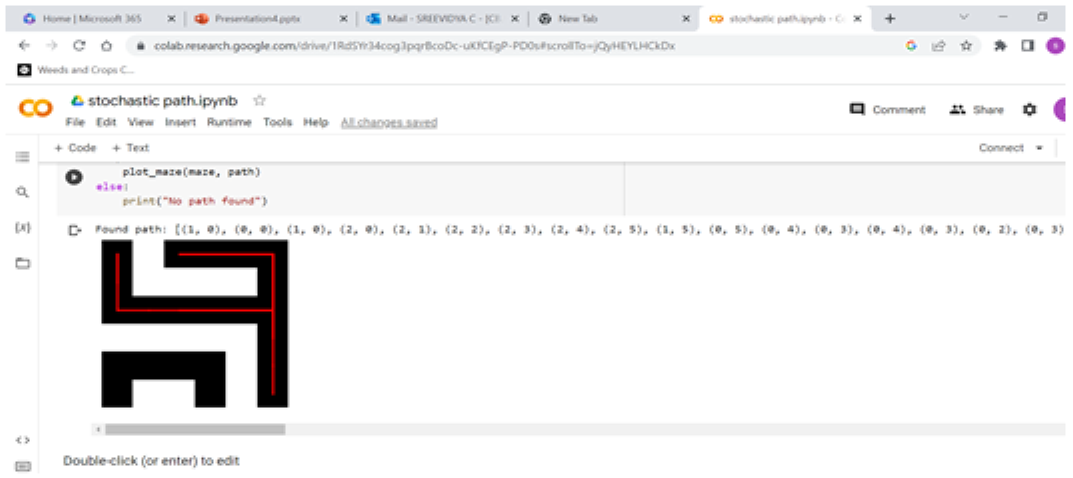


```
4.959841316620341
(-0.5, 5.5, 5.5, -0.5)
```

```
[ ]  # plot nodes
```



```
[ ] def plot_maze(maze, path=None):
        fig, ax = plt.subplots()
        rows, cols = len(maze), len(maze[0])
        ax.imshow(np.array(maze), cmap='gray')
        if path:
            for i in range(len(path) - 1):
                start = path[i]
                stop = path[i+1]
                ax.plot([start[1], stop[1]], [start[0], stop[0]], 'r-', linewidth=2.5)
        ax.axis('off')
        plt.show()

    def stochastic_shortest_path(maze, start, goal):
        rows, cols = len(maze), len(maze[0])
        state = start
        path = []

        while state != goal:
            row, col = state
            neighbors = [(row-1, col), (row, col-1), (row, col+1), (row+1, col)]
            valid_neighbors = [(r, c) for r, c in neighbors if 0 <= r < rows and 0 <= c < cols and maze[r][c] == 0]
            if valid_neighbors:
                state = random.choice(valid_neighbors)
```



Home | Microsoft 365
Presentation4.pptx
Mail - SREEVIDYA C - [CI
New Tab
stochastic path.ipynb - C
colab.research.google.com/drive/1RdSYr34cog3pqrBcoDc-uKfCEgP-PD0s#scrollTo=jQyHEYLHCkDx
Weeds and Crops C...
stochastic path.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
Connect
Double-click (or enter) to edit

```
        plot_maze(maze, path)
    else:
        print("No path found")

Found path: [(1, 0), (0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (1, 5), (0, 5), (0, 4), (0, 3), (0, 4), (0, 3), (0, 2), (0, 3)
```

Shortest Path (SSP) algorithm using a Markovian Decision Rule and a Greedy policy

This function takes a maze, a start position, and a goal position as inputs. It initializes the current state to the start position and an empty path. It then enters a loop that continues until the current state is the goal position. In each iteration, it finds all the neighbors of the current state and selects one of them randomly as the next state, as long as there are valid neighbors. If there are no valid neighbors, it returns none, indicating that no path was found. The new state is then added to the path.

Thank you