AI Assignment 1

The maze generation code has been provided by Github user kunaltyagi760 link

Maze sizes considered for comparisons: 10x10 20x20 and 25x25.

I tried maze sizes ranging from 5x5 up to 100x100. However after crossing 25x25, it was evident that the performance of all 5 algorithms will be more or less similar where the MDP algorithms particularly MDP Policy iteration was taking longer to solve the maze.

Justification of design choices

(a) Heuristic for A*

Manhattan distance calculates the sum of the absolute differences in row and column coordinates, and since for this maze only vertical and horizontal movements are allowed i.e. why I have chosen this heuristic function since it prioritises A* to avoid unnecessary paths and follow the shortest one and is computationally less expensive

(b) MDP Parameters

MDP Policy Iteration

(i) Discount Factor (discount_factor): 0.9

The discount factor of 0.9 in the code balances the algorithm's focus on immediate rewards and the consideration of future rewards when making decisions within the maze environment. It encourages a strategy that prioritizes reaching the goal quickly while still acknowledging the value of future rewards.

(ii) Policy initiation (policy): Random actions

Ensures the algorithm initially explores different parts of the maze and interacts with various states and actions before deciding on a path.

MDP Value Iteration

(i) Convergence Threshold (delta): 0.0001 or 1e-4

Calculates the absolute difference between the old value and the updated value for each state during an iteration. A low delta value means the maximum change between iterations is now so minimal that an optimal path is now likely found and breaks the loop

(ii) Reward for goal state: 1

For all other states the reward is 0 while for reaching goal state (end state) reward is 1 so that the iteration can stop when it reaches the end state.

(c) Comparison Criteria: Time complexity and Space Complexity

Both are standard comparison criterias as they are the deciding parameters when choosing an algorithm since fast and efficient algorithms are always preferred first. I have used the time and psutils library for calculating runtime and memory consumed respectively.

Below are the time and memory values obtained by running each of the algorithm for specific maze sizes.

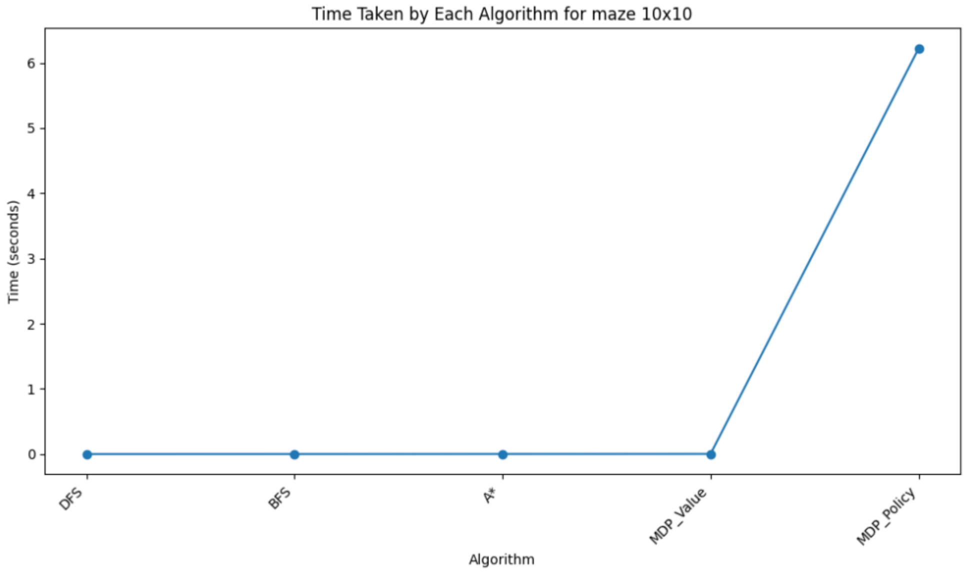
Run Time (in seconds)

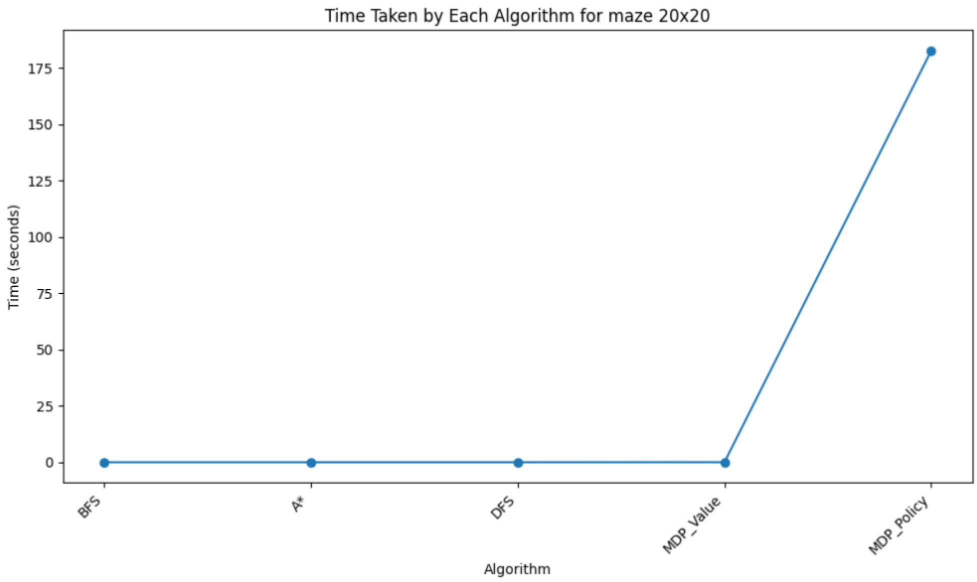
	Maze Size (nxn)		
Algorithm	10x10	20x20	25x25
DFS	0.0004	0.0024	0.0028
BFS	0.0008	0.0007	0.0009
A*	0.0018	0.0010	0.007
MDP Value	0.0027	0.0208	0.0343
MDP Policy	6.219	182.7114	521.1964

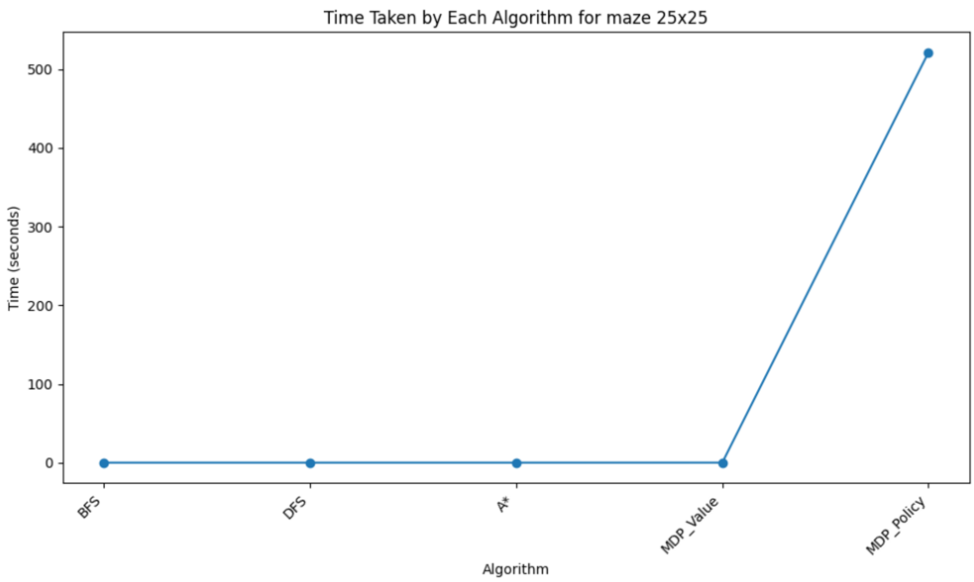
Space Consumed (in MB)

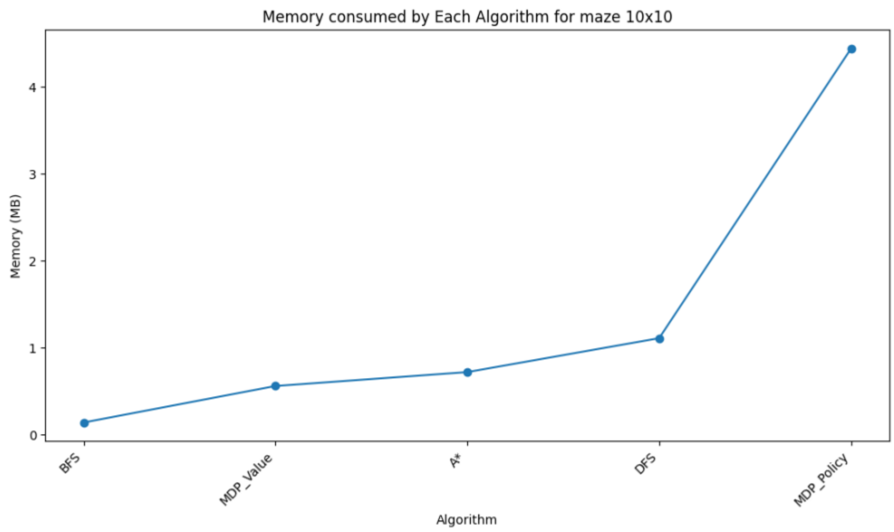
	Maze Size (nxn)		
Algorithm	10x10	20x20	25x25
DFS	1.11	1.98	2.70
BFS	0.14	0.08	0.14
A*	0.72	0.83	0.86
MDP Value	0.56	0.5	0.61
MDP Policy	4.44	6.52	6.14

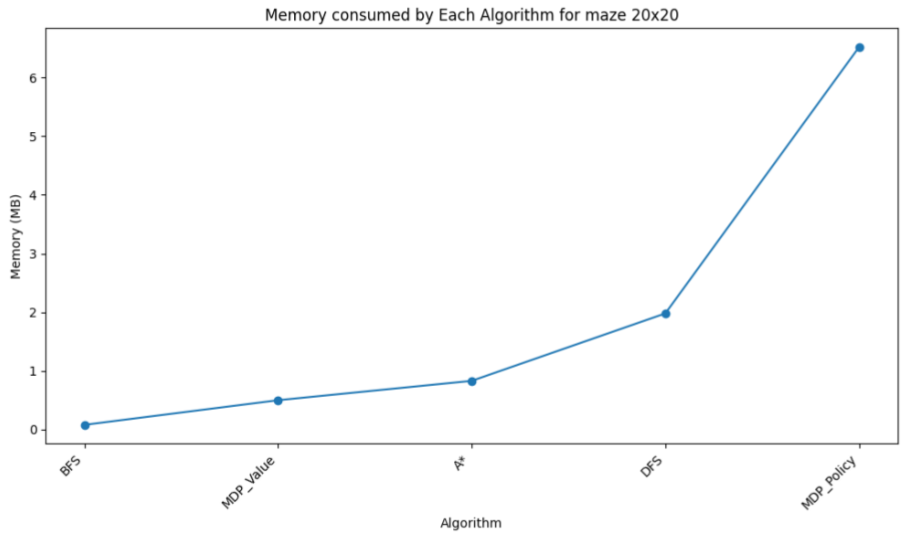
For generating plots for comparisons, I am using matplotlib library.
Below are the comparison line plots for the above tables

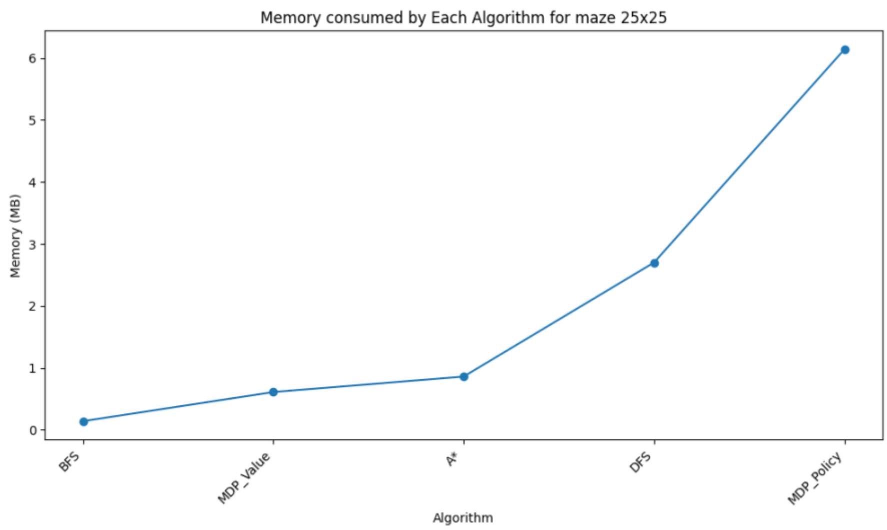

Time Taken by Each Algorithm for maze 10x10
Time (seconds)
0
1
2
3
4
5
6
DFS
BFS
A*
MDP_Value
MDP_Policy
Algorithm


Time Taken by Each Algorithm for maze 20x20
Time (seconds)
0
25
50
75
100
125
150
175
BFS
A*
DFS
MDP_Value
MDP_Policy
Algorithm


Time Taken by Each Algorithm for maze 25x25
Time (seconds)
0
100
200
300
400
500
BFS
DFS
A*
MDP_Value
MDP_Policy
Algorithm


Memory consumed by Each Algorithm for maze 10x10
Memory (MB)
0
1
2
3
4
BFS
MDP_Value
A*
DFS
MDP_Policy
Algorithm


Memory consumed by Each Algorithm for maze 20x20
Memory (MB)
0
1
2
3
4
5
6
BFS
MDP_Value
A*
DFS
MDP_Policy
Algorithm


Memory consumed by Each Algorithm for maze 25x25
Memory (MB)
0
1
2
3
4
5
6
BFS
MDP_Value
A*
DFS
MDP_Policy
Algorithm

Comparison between DFS, BFS and A*

From the above plots we can deduce that overall BFS consumes the least memory and takes least amount of time to reach the end state because DFS and A* might explore unnecessary paths before finding shortest paths meanwhile BFS works on exploring possibilities row by row or level by level. The choice of heuristic function and the complexity and size of the maze do play their part as well.
When it comes to space consumption since A* must store path plus heuristic data it consumes more amount of storage.

Comparison between MDP Value and Policy Iteration

Value iteration not only consumes less memory but is also more efficient than Policy iteration because:

- Policy iteration needs to store both policy and value function so consumes more storage.
- Value iteration function converges faster since at every step policy iteration function looks to improve its policy first.

Comparison between MDP iterations vs BFS, DFS and A*

From the table and graphs, although the difference between BFS, DFS and A* isn't substantial it is still evident BFS is the fastest and MDP Policy takes more and more time as the maze gets bigger and more complex the policy function may take an undesirable path and get stuck in a loop. In general trends also BFS, DFS and A* appear to take less time than MDP value and MDP policy iterations.
Although MDP value takes less memory compared to DFS and A* however the difference isn't substantial enough compared to MDP Policy.

Appendix

Assign.py
#Maze generation code
#https://github.com/kunaltyagi760/Terminal_based_Maze_Solver/blob/main/solve_maze.py

```
import copy
import random
import time
from a_star import find_path_astar
from bfs import find_path_bfs
from dfs import find_path_dfs
from mdp_value import find_path_value_iteration
from mdp_policy import find_path_policy_iteration
import psutil
# Constants for maze characters
WALL = '\033[91m█\033[0m'  # Red color
OPEN_SPACE = '\033[94m◌\033[0m'  # Blue color
START = '\033[92mS\033[0m'  # Green color
END = '\033[92mE\033[0m'  # Green color
PATH = '\033[92m◍\033[0m'  # Green color

def generate_maze(n, wall_percentage=25):

    # Generates a random maze of size n x n with walls and open spaces.

    # Parameters:
    #  n: Size of the maze.
    #  wall_percentage: Percentage of walls in the maze.

    # Returns:
    #  The generated maze as a 2D list.

    maze = [[OPEN_SPACE] * n for _ in range(n)]

    # Add walls
    num_walls = int((wall_percentage / 100) * (n * n))
    for _ in range(num_walls):

        row, col = random.randint(0, n - 1), random.randint(0, n - 1)
        maze[row][col] = WALL

    # Set start and end points
    maze[0][0] = START
    maze[n - 1][n - 1] = END

    return maze
```

```
def print_maze(maze):

    # Prints the maze in the terminal.

    # Parameters:
    #  maze: The maze to be printed.

    for row in maze:

        # colored_str string is printed before and after every row of maze to enhance maze
representation in terminal alse maze cell is clearly visible.

        colored_str = (('\033[91m' + "+---" + '\033[0m') * len(maze)) + '\033[91m' + "+" +
'\033[0m'
        print(colored_str)
        for cell in row:
            print("|", end=" ")
            print(cell, end=" ")
        print("|", end=" ")
        print()
    print(colored_str)

def main():
        while True:
            try:
                n = int(input("Enter the size of the maze (n x n): "))
                if n <= 1:
                    raise ValueError
                break
            except ValueError:
                print("\nInvalid input, Please enter the size of maze (n x n) > 1")

        maze = generate_maze(n)
        print("\nGenerated Maze:")
        print_maze(maze)
        gen_maze = copy.deepcopy(maze)

        while True:
            user_choice = input("\n1. Print the BFS path\n2. Print DFS path\n3. Print A*\n4. Print
the Value iteration path\n5. Print the Value Policy path\nEnter your choice(1/2/3/4/5): ")

            try:
                path, memory_usage, total_time = find_path_and_measure_memory(user_choice,
gen_maze, n)
                if path is None:
                    print(f"\nNo path found for {user_choice}")
                    continue

                if path:
```

```
                mark_path(gen_maze, path, PATH)
                print(f"\nMaze with {user_choice} Path:")
                print_maze(gen_maze)
            else:
                print(f"\nNo path found for {user_choice}")

            print(f"Time taken: {total_time:.4f} seconds")
            print(f"Memory usage: {memory_usage:.2f} MB")
        except Exception as e:  # Catch any unexpected errors
            print(f"An error occurred: {e}")

    # Allow user to continue or exit
        choice = input("\nDo you want to try another algorithm? (y/n): ")
        if choice.lower() != 'y':
            break
def mark_path(maze, path, char):
    for row, col in path:
        maze[row][col] = char

def find_path_and_measure_memory(algo_choice, maze, n):

    process = psutil.Process()
    start_memory = process.memory_info().rss / 1024 / 1024  # Initial memory usage (MB)

    start_time = time.time()
    if algo_choice == '1':
        path = find_path_bfs(maze, 0, 0, n - 1, n - 1)
    elif algo_choice == '2':
        path = find_path_dfs(maze, 0, 0, n - 1, n - 1)
    elif algo_choice == '3':
        path=find_path_astar(maze, 0, 0, n - 1, n - 1,n)
    elif algo_choice == '4':
        path=find_path_value_iteration(maze,n)
    elif algo_choice == '5':
        path=find_path_policy_iteration(maze,n)
    else:
        print(f"Invalid algorithm choice: {algo_choice}")
        return None, None

    end_memory = process.memory_info().rss / 1024 / 1024  # Memory usage after
pathfinding
    end_time = time.time()
    total_time = end_time - start_time
    memory_usage = end_memory - start_memory

    return path, memory_usage, total_time
main()
```

---

a_star.py

```
import heapq
```

```
WALL = '\033[91m█\033[0m'  # Red color
OPEN_SPACE = '\033[94m◌\033[0m'  # Blue color
START = '\033[92mS\033[0m'  # Green color
END = '\033[92mE\033[0m'  # Green color
PATH = '\033[92m◍\033[0m'  # Green color

def find_path_astar(maze, start_row, start_col, end_row, end_col, size):
    # Heuristic function for Manhattan distance
    def heuristic(row, col):
        return abs(row - end_row) + abs(col - end_col)

    open_set = [(0, heuristic(start_row, start_col), start_row, start_col)]  # Priority queue
    came_from = {}  # Store predecessors for path reconstruction
    g_score = {(start_row, start_col): 0}  # Cost from start to current cell
    f_score = {(start_row, start_col): heuristic(start_row, start_col)}  # Total estimated cost

    while open_set:
        current_f_score, _, current_row, current_col = heapq.heappop(open_set)

        if (current_row, current_col) == (end_row, end_col):
            # Reconstruct the path
            path = []
            while (current_row, current_col) in came_from:
                path.append((current_row, current_col))
                current_row, current_col = came_from[(current_row, current_col)]
            path.reverse()  # Start from the beginning
            return path

        for neighbor_row, neighbor_col in [(current_row - 1, current_col), (current_row + 1,
current_col),
                                          (current_row, current_col - 1), (current_row, current_col + 1)]:
            if 0 <= neighbor_row < size and 0 <= neighbor_col < size and
maze[neighbor_row][neighbor_col] != WALL and (neighbor_row, neighbor_col) not in
came_from:
                tentative_g_score = g_score[(current_row, current_col)] + 1
                if (neighbor_row, neighbor_col) not in g_score or tentative_g_score <
g_score[(neighbor_row, neighbor_col)]:
                    came_from[(neighbor_row, neighbor_col)] = (current_row, current_col)
                    g_score[(neighbor_row, neighbor_col)] = tentative_g_score
                    f_score[(neighbor_row, neighbor_col)] = tentative_g_score +
heuristic(neighbor_row, neighbor_col)
                    heapq.heappush(open_set, (f_score[(neighbor_row, neighbor_col)],
heuristic(neighbor_row, neighbor_col), neighbor_row, neighbor_col))

    return None  # No path found

import collections

WALL = '\033[91m█\033[0m'  # Red color
OPEN_SPACE = '\033[94m◌\033[0m'  # Blue color
```

```
START = '\033[92mS\033[0m'  # Green color
END = '\033[92mE\033[0m'  # Green color
PATH = '\033[92m◍\033[0m'  # Green color
```

Bfs.py

```
def find_path_bfs(maze, start_row, start_col, end_row, end_col):

    n = len(maze)
    visited = [[False] * n for _ in range(n)]
    queue = collections.deque([(start_row, start_col, [])])

    while queue:
        cur_row, cur_col, path = queue.popleft()

        if cur_row == end_row and cur_col == end_col:
            return path + [(cur_row, cur_col)]

        if visited[cur_row][cur_col]:
            continue

        visited[cur_row][cur_col] = True

        for dr, dc in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            new_row, new_col = cur_row + dr, cur_col + dc
            if 0 <= new_row < n and 0 <= new_col < n and maze[new_row][new_col] != WALL:
                queue.append((new_row, new_col, path + [(cur_row, cur_col)]))

    return None  # No path found
```

Mdp_value.py

```
WALL = '\033[91m█\033[0m'  # Red color
OPEN_SPACE = '\033[94m◌\033[0m'  # Blue color
START = '\033[92mS\033[0m'  # Green color
END = '\033[92mE\033[0m'  # Green color
PATH = '\033[92m◍\033[0m'  # Green color

def find_path_value_iteration(maze, size):
    # Initialize values and state-action pairs
    values = [[0] * size for _ in range(size)]
    actions = [[None] * size for _ in range(size)]
    values[size - 1][size - 1] = 1  # Set end point value to 1

    # Define actions (up, down, left, right)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Value iteration loop
    while True:
        delta = 0
        for row in range(size):
            for col in range(size):
```

```
                if maze[row][col] == WALL or (row == size - 1 and col == size - 1):
                    continue  # Skip walls and end point

                current_value = values[row][col]
                best_value = float('-inf')
                best_action = None

                for direction in directions:
                    new_row, new_col = row + direction[0], col + direction[1]
                    if 0 <= new_row < size and 0 <= new_col < size and maze[new_row][new_col] != WALL:
                        reward = -1  # Assume a negative step cost for moving
                        new_value = values[new_row][new_col] + reward
                        if new_value > best_value:
                            best_value = new_value
                            best_action = direction

                values[row][col] = best_value
                actions[row][col] = best_action
                delta = max(delta, abs(current_value - values[row][col]))

        if delta < 1e-4:  # Convergence threshold
            break

    # Trace the path from start to end
    path = []

    row, col = 0, 0
    while row != size - 1 or col != size - 1:
        action = actions[row][col]
        maze[row][col] = PATH  # Mark the path
        path.append((row, col))
        row += action[0]
        col += action[1]

    maze[size - 1][size - 1] = END  # Mark the end point
    return path
```

Dfs.py

```
WALL = '\033[91m█\033[0m'  # Red color
OPEN_SPACE = '\033[94m◌\033[0m'  # Blue color
START = '\033[92mS\033[0m'  # Green color
END = '\033[92mE\033[0m'  # Green color
PATH = '\033[92m◍\033[0m'  # Green color

def find_path_dfs(maze, start_row, start_col, end_row, end_col):

    n = len(maze)
    visited = [[False] * n for _ in range(n)]
    stack = [(start_row, start_col, [])]
```

```
    while stack:
        cur_row, cur_col, path = stack.pop()

        if cur_row == end_row and cur_col == end_col:
            return path + [(cur_row, cur_col)]

        if visited[cur_row][cur_col]:
            continue

        visited[cur_row][cur_col] = True

        for dr, dc in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            new_row, new_col = cur_row + dr, cur_col + dc
            if 0 <= new_row < n and 0 <= new_col < n and maze[new_row][new_col] != WALL:
                stack.append((new_row, new_col, path + [(cur_row, cur_col)]))

    return None  # No path found
import heapq
```

Mdp_policy.py

```
import numpy as np
WALL = '\033[91m█\033[0m'  # Red color
OPEN_SPACE = '\033[94m◌\033[0m'  # Blue color
START = '\033[92mS\033[0m'  # Green color
END = '\033[92mE\033[0m'  # Green color
PATH = '\033[92m◍\033[0m'  # Green color

def find_path_policy_iteration(maze, size):
    # Define MDP components
    states = [(row, col) for row in range(size) for col in range(size)]
    actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Up, right, down, left
    rewards = np.zeros((size, size))
    rewards[size - 1, size - 1] = 1  # Reward at the goal state
    discount_factor = 0.9

    # Define transition probabilities (adjust for walls and boundaries)
    transition_probabilities = np.zeros((size, size, len(actions), size, size))
    for row in range(size):
        for col in range(size):
            for action_index, action in enumerate(actions):
                new_row = min(size - 1, max(0, row + action[0]))
                new_col = min(size - 1, max(0, col + action[1]))

                if maze[new_row][new_col] != WALL:
                    transition_probabilities[row, col, action_index, new_row, new_col] = 1

    # Initialize policy (random actions)
    policy = np.random.choice(len(actions), size * size).reshape(size, size)
```

```
    # Policy iteration loop
    while True:
        # Policy evaluation
        value_function = np.zeros((size, size))
        policy_stable = True

        while True:
            new_value_function = np.zeros((size, size))
            for row in range(size):
                for col in range(size):
                    action_index = policy[row, col]
                    for new_row in range(size):
                        for new_col in range(size):
                            transition_prob = transition_probabilities[row, col, action_index, new_row, new_col]
                            new_value_function[row, col] += transition_prob * (rewards[new_row, new_col] + discount_factor * value_function[new_row, new_col])

            if np.allclose(value_function, new_value_function):
                break

            value_function = new_value_function

        # Policy improvement
        policy_stable = True
        for row in range(size):
            for col in range(size):
                best_action_index = None
                best_action_value = float('-inf')
                for action_index in range(len(actions)):
                    action_value = 0
                    for new_row in range(size):
                        for new_col in range(size):
                            transition_prob = transition_probabilities[row, col, action_index, new_row, new_col]
                            action_value += transition_prob * (rewards[new_row, new_col] + discount_factor * value_function[new_row, new_col])

                    if action_value > best_action_value:
                        best_action_index = action_index
                        best_action_value = action_value

                if policy[row, col] != best_action_index:
                    policy_stable = False
                    policy[row, col] = best_action_index

        if policy_stable:
            break

    # Extract path from policy
```

```
    path = []
    row, col = 0, 0
    while (row, col) != (size - 1, size - 1):
        path.append((row, col))
        action = actions[policy[row, col]]
        row += action[0]
        col += action[1]

    return path
```

Plot.py

```
import matplotlib.pyplot as plt
#algorithms={'BFS':0.0004,'DFS':0.0002,'A*':0.0003,'MDP_Value':0.0009,'MDP_Policy':0.2636} #5X5 maze
#algorithms={'BFS':0.0008,'DFS':0.0004,'A*':0.0018,'MDP_Value':0.0027,'MDP_Policy':6.219} #10x10 maze
#algorithms={'BFS':0.0007,'DFS':0.0024,'A*':0.0010,'MDP_Value':0.0208,'MDP_Policy':182.7114 } #20x20 maze
algorithms={'BFS':0.0009,'DFS':0.0028,'A*':0.007,'MDP_Value':0.0343,'MDP_Policy':521.1964} #25x25 maze
sorted_algorithms = sorted(algorithms, key=algorithms.get)

# Extract sorted algorithms and times
algorithms_list = list(sorted_algorithms)
times_list = [algorithms[algorithm] for algorithm in algorithms_list]

plt.figure(figsize=(10, 6))  # Set plot dimensions
plt.plot(algorithms_list, times_list, marker='o', linestyle='-')  # Plot line with markers
plt.xlabel("Algorithm")
plt.ylabel("Time (seconds)")
plt.title("Time Taken by Each Algorithm for maze 50x50")
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better visibility
plt.tight_layout()
plt.show()
```

Plot_memory.py

```
import matplotlib.pyplot as plt
#algorithms={'BFS':0.14,'DFS':1.11,'A*':0.72,'MDP_Value':0.56,'MDP_Policy':4.44} #10x10 maze
algorithms={'BFS':0.08,'DFS':1.98,'A*':0.83,'MDP_Value':0.5,'MDP_Policy':6.52} #20x20 maze
#algorithms={'BFS':0.14,'DFS':2.70,'A*':0.86,'MDP_Value':0.61,'MDP_Policy':6.14} #25x25 maze
sorted_algorithms = sorted(algorithms, key=algorithms.get)

# Extract sorted algorithms and times
algorithms_list = list(sorted_algorithms)
memory_list = [algorithms[algorithm] for algorithm in algorithms_list]
```

```
plt.figure(figsize=(10, 6))  # Set plot dimensions
plt.plot(algorithms_list, memory_list, marker='o', linestyle='-')  # Plot line with markers
plt.xlabel("Algorithm")
plt.ylabel("Memory (MB)")
plt.title("Memory consumed by Each Algorithm for maze 50x50")
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better visibility
plt.tight_layout()
plt.show()
```