AI Assignment 1

The maze generation code has been provided by Github user kunaltyagi760 [link](https://github.com/kunaltyagi760/Terminal_based_Maze_Solver/blob/main/solve_maze.py)

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**

1. 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

1. MDP Parameters

MDP Policy Iteration

1. 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.

1. 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

1. 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

1. 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.

1. 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

A graph with a blue line

Description automatically generatedA graph with a line

Description automatically generatedA graph with a line

Description automatically generated

A graph with a line

Description automatically generated

A graph with a line

Description automatically generatedA graph with a line

Description automatically generated

**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()