CS7IS2 Artificial Intelligence

Assignment 1 – Maze Search

Name: Sai Bala Subrahmanya Lakshmi Kanth Rayanapati

Student ID: 19304511

Link to the Demo can be found here

INTRODUCTION

The main goal of this assignment is the generate a maze of different sizes, design and complexity, and search these mazes employing a suite of search algorithms including Breadth First Search (BFS), Depth First Search (DFS), A*, and Markov Decision Processes (MDPs) with Value Iteration and Policy Iteration techniques. The primary objective is to thoroughly assess and compare these algorithms across various performance metrics: shortest path, number of cells explored, time efficiency and memory usage to evaluate their ability to scale to various complex mazes. This study provides a comprehensive evaluation of the performance characteristics, limitations of different algorithms in solving the maze.

IMPLEMENTATION

Maze Generation through Pyamaze:

For the task of generating mazes of varying sizes, designs and complexities, I used **Pyamaze** (License), an open-source library dedicated for maze generation. This library is highly customisable, allowing us to customise mazes by specifying *dimensions* and various other attributes through its *CreateMaze()* function. Other customisable attributes used in this assignment are *loopPercent*, which changes the density of paths within the maze, and *pattern*, that allows us to select none, horizontal, or vertical patterns. Additionally, Pyamaze offers the capability to save a generated maze in CSV format which can be later loaded as a maze using *CreateMaze()* function. This feature was very useful at the start of the project to evaluate the performance of different algorithms against the same maze. *MazeMap()* is another useful function that provides the information of all the cells and their surrounding walls. Further, Pyamaze also has inbuilt functions to visualise the maze and the agent that searches the maze.

For this assignment I leveraged Pyamaze to generate mazes of varying sizes: 5x5, 10x10, 20x20, 50x50, and 100x100, distinct patterns (none, horizontal, and vertical) and loop percentages (0%, 50%, and 100%). All the mazes generated were solved using all five algorithms: Breadth First Search (BFS), Depth First Search (DFS), A*, Value Iteration, and Policy Iteration to gain insights into the impact of different maze characteristics.

Search Algorithms

Breadth-First Search (BFS)

In my implementation of the Breadth-First Search (BFS) algorithm, the focus is on identifying the shortest path from the start cell to the goal cell to solve the maze. Start cell is the maze's bottom-right corner cell of the maze, designated as (*maze.rows, maze.cols*), and the goal cell is the top-left corner cell, marked as (1,1).

The *bfs()* function take a maze object as input that contains detailed information about the maze's layout, including its cells and the walls that define possible paths. As the algorithm progresses, it explores all the adjacent cells to the current cell by avoiding any cell that it has previously explored. This exploration is done by employing a queue data structure(First In, First Out) to ensure that the exploration is done in a breadth-first manner. It explores all the neighbours of the current cell before progressing to a deeper level.

The algorithm explores the maze and stores the explored path in a key-value pair format. This is later used to back-track the path to find the shortest path between the start and the goal cell.

Depth-First Search (DFS)

The implemented Depth-First Search (DFS) algorithm finds the path between the staring cell and the goal cell to solve the maze. Start cell is the maze's bottom-right corner cell of the maze, designated as (*maze.rows, maze.cols*), and the goal cell is the top-left corner cell, marked as (1,1).

The *dfs()* function take a maze object as input that contains detailed information about the maze's layout, including its cells and the walls that define possible paths. In contrast to the BFS algorithm, DFS explores the depth of each branch before considering alternatives following the directional order of preference West->North->South->East while avoiding the previously visited cells. This explorations is done using a Stack data structure (Last In, First Out) to ensure that the exploration is done in depth-first manner.

Further, the algorithm explores the maze in a depth-first manner and stores the explored path in a key-value pair format. This is later used to back-track the path to reconstruct the shortest path between the start and the goal cell.

A* Search

In the implementation of the A* algorithm, I defined a heuristic function(*h_score*()) that takes 2 cells as input and estimates the distance between the two cells by employing Manhattan distance. In the case of this assignment, the first cell is the current cell and second is the goal cell. *h_score()* provides the estimate for the cheapest path from the current cell to the goal cell. The algorithm also utilises *g_score()* to track the cost of the shortest path from the start cell to the current cell. The *g_score* value was initially set to infinity for all the cells. The *f_score()* is the total estimated cost to reach the goal cell from the current cell. It is calculated by *adding* the *h_score* and *g_score* values for any cell. The *f_score* values are initially set the infinity and are updated as the algorithm explores the maze.

This algorithm employs a priority queue data structure to manage the exploration ensuring that the cells are explored in a manner that prioritises the cells with the lowest *f_score.* Upon selecting and exploring the cell with the least *f_score,* the algorithm evaluates its neighbours in all the possible directs and updates their *f_score* if the newly evaluated *f_score* for the cell is lower than the previous *f_score.* These cells are then added to the priority que for further exploration. This approach is continued until the goal cell is reached.

Upon reaching the goal cell, the algorithm back-tracks the explored path dictionary from goal cell to the start cell to construct the path.

Markov Decision Processes (MDPs)

In the context of solving a maze using Markov Decision Processes (MDPs), the maze itself is modelled as the MDP environment. Each cell within the maze is treated as a state, with movements in the directions of East, North, South, and West representing the available actions.

Policy Iteration

The *initialisation()* function sets up the MDP framework by defining the cells as the various states, assigning rewards all the states with the goal state receiving a reward of 1 and all the other states receiving a negative reward of -0.04 and initialising the value function of each state 0. The transition probabilities are defined with a 90% chance of the action resulting in a change of state and a 10% chance that the state will remain the same. For each state the initial policy is chosen at random from the possible actions from the current state with a probability of 1. Further, the hyperparameters are set with the discount factor (gamma) equal to 0.99 and threshold (theta) equal to 0.001.

The core algorithm operates by evaluating the current policy and improving it iteratively. In policy evaluation, we calculate the value function (v) of each state under the current policy by considering the rewards and the transition probabilities to other states. The value function is calculated and updated using the Bellman equation:

$$V(s) = \sum_{s'} \boldsymbol{T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V(s')]}$$

Where:

- V(s) is the value function of state s
- $T(s,\pi(s),s')$ is the transition probability in transiting from state s to state s' as dictated by policy π(s)
- R(s,π(s),s') is the reward received upon transitioning from state s to s' as directed by policy π(s)
- γ is the discount factor
- V(s') is the value of the subsequent state s'

This process iterates over all the states and updates their value functions by following the current policy.

Subsequently, in policy improvement phase, the algorithm is revised to be more "greedy" and choose the actions that maximise the value function based on the current policy.

Finally, the path for the maze from the start cell to the goal cell is derived by choosing the actions directed by the optimal policy π for each state.

Value Iteration

Similar to policy evaluation, the states, rewards, initial policies, value functions and the threshold values are initialised in the *initialisation()* function.

In the implementation, the algorithm iteratively updates the value of each state until the delta is less than the threshold value. For each state, it considers all the possible actions and updates the value to the max expected value obtainable from the actions. This is done using the Bellman equation:

$$V(s) = max_a \sum_{s'} \boldsymbol{R(s,\pi(s),s') + \gamma V(s')}$$

Finally, to get the optimal path *getPath()* function traces the optimal path from the start to the goal by following the actions prescribed by the optimal policy, similar to the approach in Policy Iteration but directly informed by the value function refined through Value Iteration.

EVALUATION

Comparison between different search algorithms

For evaluating and comparing the performance of various search algorithms, I conducted evaluations on mazes of differing sizes, patterns and loop percentages. The key performance metrics like the length of the shortest path, the number of cells explored, execution time, and memory consumption were measured for analysis. The findings are presented bellow in a series of tables. All the data present in the tables are the average values obtained after 10 iterations on mazes of different sizes, patterns and loop sizes.

Maze Size Variation

To evaluate the performance of different search algorithms, the first comparison is done on mazes of different dimensions. Here the mazes of varying sizes were assessed to evaluate the scalability and efficiency of different search algorithms while the other parameters (Pattern and loop percent) are kept constant.

The mazes of sizes: 5x5, 10x10, 20x20, 50x50 and 100x100 were used while maintaining a consistent loop percent (50%) and pattern (none).

(5x5 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	9	25	0.00007309	0.00078125
DFS	11	16.1	0.00003794	0
A* Search	9	11.4	0.00017094	0.00078125

Table 1 – Average values of 10 iterations for maze of size 5x5

(10x10 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	21	97.9	0.00032985	0.000390625
DFS	24.6	37.2	0.00008814	0
A* Search	21	37.7	0.0004709	0.000390625

Table 2 – Average values of 10 iterations for maze of size 10x10

(20x20 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	41.6	399.6	0.0035957	0.00078125
DFS	68.6	116.3	0.0004441	0.00625
A* Search	41.6	173.4	0.00228514	0.002734375

Table 3 – Average values of 10 iterations for maze of size 20x20

(50x50 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	104.2	2498.8	0.11684054	0.083984375
DFS	168.6	396.3	0.00370779	0.000390625
A* Search	104.2	848.4	0.01481336	0.03359375

Table 4 – Average values of 10 iterations for maze of size 50x50

(100x100 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	209.8	9999	3.80461715	0.368359375
DFS	345	1111.3	0.05258333	0.0015625
A Star Search	209.8	3722.2	0.4069423	0.21015625

Table 5 – Average values of 10 iterations for maze of size 100x100

Key Observations

The key observations from the above results are:

- Both BFS and A* consistently find the optimal shortest solution even as the maze size increases.
- DFS on average always has a higher number of cells (longer paths) in its solution when compared to the other algorithms.
- BFS finds the optimal shortest path in all the mazes but is not scalable larger mazes as it explores almost all the cells in a maze before finding the shortest path. In contrast, DFS has the least number of cells explored but doesn't find the shortest path.
- A* generally finds the shortest path and explores significantly fewer cells compared to BFS making it good for scaling to larger mazes.
- The time efficiency varies significantly with the maze size. DFS is usually the fastest algorithm of the three, followed by A* with BFS lagging. BFS takes significantly more time when compared to the other 2 algorithms.
- The memory usage of DFS is much lower when compared to the other two algorithms. This is followed by A* and then BFS.
- Overall, A* algorithm scales well to mazes of all sizes as it always finds shortest path of the maze and explores significantly lower cells when compared to BFS.

Loop Percent Variation

In this case, we evaluate the algorithms' performance on the mazes of the size 100x100, varying loop percentages from 0% (representing a perfect maze with a single solution) to 100% (Multiple solutions). This variation allows us to examine the algorithms adaptability and efficiency in scenarios where multiple paths are present.

The mazes of size: 100x100 with no pattern were used while varying the loop percent (100%, 50% and 0%).

(0% loop)	Shortest Path	Cells Explored	Time Taken (s)	Memory Usage (MB)
BFS	2502.6	8755.3	1.400357	0.365625
DFS	2502.6	7688.7	1.010213	0.012891
A* Search	2502.6	8633.7	0.908514	0.198438

Table 6 – Average values of 10 iterations for maze of 0% loop

(50 % loop)	Shortest Path	Cells Explored	Time Taken (s)	Memory Usage (MB)
BFS	210.8	9999.4	1.921681	0.474219
DFS	455.6	1414.8	0.049133	0.002344
A* Search	210.8	3961	0.230609	0.092969

Table 7 – Average values of 10 iterations for maze of 50% loop

(100 % loop)	Shortest Path	Cells Explored	Time Taken (s)	Memory Usage (MB)
BFS	198	10000	2.27984	0.417969
DFS	231.6	406.4	0.002738	0
A* Search	198	1017.5	0.024317	0

Table 8 – Average values of 10 iterations for maze of 100% loop

Key Observations

Some key observations are:

- DFS is the best performing algorithm in a perfect maze by minimising the number of cells explored and the memory consumed. Interestingly, A* outperforms DFS in terms of speed even though it explored more cells when compared to DFS.
- As expected, BFS takes the most amount of time in all the scenarios as it explores nearly all the cells in the maze.
- BFS uses significantly higher memory when compared to the other two algorithms.
- Both BFS and A* always find the shortest possible solution of the maze.
- DFS is a compelling choice when the preference for solving the maze is speed over the optimal path.
- A* is the most optimal algorithm as it consistently identifies the shortest path across varying loop percentages.

Pattern Variation

Here we explore the impact of various patterns in the maze on the performance of various algorithms.

The mazes of size: 100x100 with different patterns (no pattern, horizontal and vertical) were used while maintaining a consistent loop percent (50%)

(No pattern)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	209.6	9996.9	2.014184	0.421875
DFS	320.8	1849.1	0.202064	0.001953
A* Search	209.6	3881.1	0.224207	0.11875

Table 9 – Average values of 10 iterations for maze of no pattern

(Vertical)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	253.2	9994.3	1.891838	0.394922
DFS	350.2	536	0.004501	0
A* Search	253.2	7086.4	0.511518	0.301953

Table 10 – Average values of 10 iterations for maze of Vertical pattern

(Horizontal)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	248.8	9978.6	2.186057	0.458594
DFS	863.8	1501.2	0.079907	0
A* Search	248.8	6932.6	0.565129	0.287109

Table 11 – Average values of 10 iterations for maze of Horizontal pattern

Key Observations

The key observations are:

- Both BFS and A* always find the shortest path irrespective of the pattern in the maze.
- DFS solves the mazes faster and in lesser memory but doesn't find the shortest path.
- DFS is the most affected algorithm by differing maze patterns.
- DFS performs notably better in vertical mazes in terms of the no of cells explored resulting in finding a solution very quickly. On the other hand, it performs significantly worse in horizontal mazes.
- The no of cells explored by A* increases a notably when pattern is introduced in the maze.

- A* adapts well to mazes of all patterns and finds the most optimal path in much better time compared to BFS.
- DFS must be the choice when solution length is not important.

Overall, A* seems to be the most optimal algorithm to find the shortest possible path in the maze even though there is a slight compromise in the time taken to find the solution. However, if finding the shortest path is not the priority, DFS can be used to solve the maze as it finds the solution in the least possible time with the least memory usage.

Comparison between different MDP algorithms

Similar to the comparison of the search algorithms, the performance analysis of the MDP algorithms is done across 3 groups: varying the maze sizes, varying patterns and varying loop percentages. Shortest path, time taken, and memory used are the performance metrics on which these algorithms are evaluated. The findings are presented in the series of tables below. All the data present in the tables are the average values obtained after 10 iterations on mazes of different sizes, patterns and loop sizes.

Maze Size Variation

The mazes of sizes: 5x5, 10x10, 20x20, 50x50 and 100x100 were used while maintaining a consistent loop percent (50%) and pattern (none).

(5x5 Maze)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	9.2	0.00948751	0.01328125
Value Iteration	9.2	0.00163007	0

Table 12 – Average values of 10 iterations for maze of size 5x5

(10x10 Maze)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	19.4	0.03974226	0.0171875
Value Iteration	19.4	0.00811152	0

Table 13 – Average values of 10 iterations for maze of size 10x10

(20x20 Maze)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	42	0.20663351	0.07578125
Value Iteration	42	0.07435077	0

Table 14 – Average values of 10 iterations for maze of size 20x20

(50x50 Maze)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	106	1.591313	0.723046875
Value Iteration	106	2.10944567	0.02265625

Table 15 – Average values of 10 iterations for maze of size 50x50

(100x100 Maze)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	212.6	11.51906534	1.8734375
Value Iteration	212.6	31.95430678	-0.08828125

Table 16 – Average values of 10 iterations for maze of size 100x100

Key Observations

Some key observations are:

- Both Policy Iteration and Value Iteration consistently find the shortest path across mazes of all sizes.

- Value iteration is quicker for mazes of smaller mazes but for larger mazes, policy iteration is significantly quicker.
- Policy iteration uses considerably more memory when compared to value iteration.
- Interestingly, the memory usage for value iteration for the 100x100 maze is negative.

Loop Percent Variation

Given the constraints of the computing resources of my machine, it was observed that value iteration is taking significantly long time to find a path when applied to a perfect maze (with loop percentage of 0) for the mazes of sizes 10x10 and larger. After running the program for a long time, it exits the program with a memory error. To accommodate these constraints and ensure a feasible evaluation of the MDPs, the decision was made to exclude the perfect maze scenario from value iteration analysis and consider a maze of size 20x20 for this evaluation.

The mazes of size: 20x20 with no pattern were used while varying the loop percent (100%, 50% and 0%).

(100% loop)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	39	0.282123	0.089453
Value Iteration	39	0.122013	0

Table 17 – Average values of 10 iterations for maze of 100% loop

(50% loop)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	41.2	0.286398	0.001953
Value Iteration	41.2	0.10462	0.000781

Table 18 – Average values of 10 iterations for maze of 50% loop

(0% loop)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	160.8	0.600248	0.013672
Value Iteration	NA	NA	NA

Table 19 – Average values of 10 iterations for maze of 0% loop

Key Observations

The key observations are:

- Both the algorithms identify the shortest path.
- Policy Iteration takes more time and memory compared to Value iteration.
- Unable to computer value iteration for a 20x20 maze with loop percent of 0%.

Pattern Variation

The mazes of size: 20x20 with different patterns (no pattern, horizontal and vertical) were used while maintaining a consistent loop percent (50%).

(No pattern)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	41.6	0.581288	0.003906
Value Iteration	41.6	0.22435	0

Table 20 – Average values of 10 iterations for maze of no pattern

(Vertical)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	44	0.847996	0.004688
Value Iteration	44	0.324305	0

Table 21 – Average values of 10 iterations for maze of vertical pattern

(Horizontal)	Shortest Path	Time Taken (s)	Memory Used (MB)
Policy Iteration	42.8	0.905957	0.003125
Value Iteration	42.8	0.322255	0

Table 22 – Average values of 10 iterations for maze of horizontal pattern

Key Observations

The key observations are:

- Both the algorithms identify the shortest path.
- Policy Iteration takes more time and memory compared to Value iteration.
- Vertical and Horizontal patterns take more time than a maze with no pattern.

Overall, both Policy Iteration and Value Iteration identifies the shortest path possible. Value Iteration is slightly faster for mazes of small sizes but is significantly slower compared to Policy Iteration in larger mazes. Also, mazes of sizes 14x14 and larger are taking unusually long time for value iteration with loop percent of 0% and exiting with a memory error.

Comparison between search and MDP algorithms

Maze Size Variation

The mazes of sizes: 5x5, 10x10, 20x20, 50x50 and 100x100 were used while maintaining a consistent loop percent (50%) and pattern (none).

(5x5 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	9.2	25	5.11E-05	0.000781
DFS	10.4	15.4	3.86E-05	0
A*	9.2	14.2	0.000181	0.000781
Policy Iteration	9.2	NA	0.014327	0.011328
Value Iteration	9.2	NA	0.00249	0

Table 23 – Average values of 10 iterations for maze of size 5x5

(10x10 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	20	99.4	0.000351	0.071875
DFS	25.2	39	9.96E-05	0
A*	20	41.9	0.000497	0.000391
Policy Iteration	20	NA	0.068536	0.006641
Value Iteration	20	NA	0.013593	0

Table 24 – Average values of 10 iterations for maze of size 10x10

(20x20 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	42.8	398.7	0.003456	0.010547
DFS	57	91.6	0.000306	0
A*	42.8	171.5	0.0018	0.000781
Policy Iteration	42.8	NA	0.296143	0.030078
Value Iteration	42.8	NA	0.112683	0.000781

Table 25 – Average values of 10 iterations for maze of size 20x20

(50x50 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	103.6	2498.6	0.282808	0.10625
DFS	190.2	312.3	0.005045	0.000391
A*	103.6	568.8	0.02237	0.001953

Policy Iteration	103.6	NA	6.811971	0.224219
Value Iteration	103.6	NA	7.502637	0.004297

Table 26 – Average values of 10 iterations for maze of size 50x50

(100x100 Maze)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	212.6	9997.8	1.728045	0.030078
DFS	336	1094.8	0.031087	0.008594
A*	212.6	5118	0.276903	0.108984
Policy Iteration	212.6	0	16.38515	0.873047
Value Iteration	212.6	0	47.06315	-0.03359

Table 27 – Average values of 10 iterations for maze of size 100x100

Key Observations

Some key observations are:

- BFS, A*, Policy Iteration and Value Iteration find the shortest path in mazes of all sizes. DFS however, results in longer paths.
- DFS and A* are notably faster when compared to BFS and the MDP algorithms.
- As the maze size increases, MDP algorithms take significantly higher time compared to search algorithms.
- Policy Iteration algorithm uses the most amount of memory.

Loop Percent Variation

The mazes of size: 20x20 with no pattern were used while varying the loop percent (100%, 50% and 0%).

(0% loop)	Shortest Path	Cells Explored	Time Taken (s)	Memory Usage (MB)
BFS	136.6	363.8	0.002497	0.002734
DFS	136.6	270.1	0.001574	0.000391
A*	136.6	343.1	0.003365	0.010938
Policy Iteration	136.6	NA	0.634885	0.048438
Value Iteration	NA	NA	NA	NA

Table 28 – Average values of 10 iterations for maze of 0% loop

(50 % loop)	Shortest Path	Cells Explored	Time Taken (s)	Memory Usage (MB)
BFS	41.4	398.9	0.003549	0.000391
DFS	51.8	101.8	0.000386	0
A*	41.4	135	0.001546	0
Policy Iteration	41.4	NA	0.306373	0.000391
Value Iteration	41.4	NA	0.110937	0

Table 29 – Average values of 10 iterations for maze of 50% loop

(100 % loop)	Shortest Path	Cells Explored	Time Taken (s)	Memory Usage (MB)
BFS	39	400	0.003723	0.002734
DFS	42.8	76.7	0.000215	0
A*	39	63.9	0.00086	0
Policy Iteration	39	NA	0.276509	0
Value Iteration	39	NA	0.123018	0

Table 30 – Average values of 10 iterations for maze of 100% loop

Key Observations

The key observations are:

- BFS, A*, Policy Iteration and Value Iteration identify the shortest path possible.
- MPD algorithms take more time to find the solution then search algorithms.

Pattern Variation

The mazes of size: 20x20 with different patterns (no pattern, horizontal and vertical) were used while maintaining a consistent loop percent (50%)

(No pattern)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	42	398.2	0.003791	0.001563
DFS	59.2	96.3	0.000353	0.00625
A*	42	181.7	0.002013	0.001953
Policy Iteration	42	NA	0.284041	0.048828
Value Iteration	42	NA	0.109679	0.000781

Table 31 – Average values of 10 iterations for maze of no pattern

(Vertical)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	45.6	399.8	0.007488	0.000781
DFS	54.2	89.4	0.00049	0
A*	45.6	226.3	0.004535	0
Policy Iteration	45.6	NA	0.629652	0.008203
Value Iteration	45.6	NA	0.253799	0.002344

Table 32 – Average values of 10 iterations for maze of Vertical pattern

(Horizontal)	Shortest Path	Cells Explored	Time Taken (s)	Memory Used (MB)
BFS	43.2	390.5	0.008988	0.000781
DFS	66.2	98.8	0.000682	0
A*	43.2	207.7	0.005026	0
Policy Iteration	43.2	NA	0.807739	0.010156
Value Iteration	43.2	NA	0.289944	0

Table 33 – Average values of 10 iterations for maze of Horizontal pattern

Key Observations

Some key observations are:

- BFS, A*, Policy Iteration and Value Iteration identify the shortest path irrespective of the maze pattern.

Overall, BFS, A*, Policy Iteration and Value Iteration identify the shortest path. MDP algorithms take longer to find the solution compared to search algorithms.

CONCLUSION

Breadth First Search (BFS) guarantees the shortest possible path but due to the exploration of almost all the cells, the time taken, and the memory usage are higher. This makes it not suitable to scale to large mazes. Depth First Search (DFS) is more memory and time efficient compared to both BFS and A* but does not identify the most optimal solution. This makes it scalable to larger mazes but only when finding the optimal path is not a priority. A* stands out as the most optimal of the search algorithms by maintaining a balance between efficiency and path optimality. Being able to find the most optimal path in significantly lesser time than BFS makes A* scalable to large mazes.

Although Policy Iteration and Value Iteration offer a different approach, they both identify the most optimal path. They consume significant time and memory resources. They are scalable to larger mazes, but I couldn't produce results due to limited computational resource availability.

In conclusion, A* emerges as the most versatile and efficient solution for solving the mazes of different sizes, patterns and complexities. It offers a balanced solution between finding the optimal path and the computational time and memory resources required.

APPENDIX

Note: The code for this project has been inspired by this tutorial and the video tutorials can be found here.

Link to my Assignment Demo video can be found here:
https://drive.google.com/drive/folders/1bWAyt3J6LjLIXxsZfTHgOe_Epf7cwpoB?usp=sharing

MazeGenerator.py

```
from pyamaze import maze


def generate_maze(x,y, loopPercent, pattern):
    m = maze(x, y)
    m.CreateMaze(loopPercent=loopPercent,pattern=pattern)
    return m

if __name__ == '__main__':
    maze = generate_maze(5,5,50,"none")
    print(maze.maze_map)
```

BreadthFirstSearch.py

```
from pyamaze import maze, agent, COLOR, textLabel
import time


def bfs(maze):
    explored_cells = []
    cells_to_explore = [(maze.rows, maze.cols)]
    explored_paths = {}

    explored_cells.append((maze.rows, maze.cols))

    while len(cells_to_explore) > 0:
        current_cell = cells_to_explore.pop(0)

        # Goal reached condition
        if current_cell == (1, 1):
            break

        for open_direction in 'ESNW':
            if maze.maze_map[current_cell][open_direction]:
                if open_direction == 'E':
                    child_cell = (current_cell[0], current_cell[1] + 1)
                elif open_direction == 'W':
                    child_cell = (current_cell[0], current_cell[1] - 1)
                elif open_direction == 'S':
                    child_cell = (current_cell[0] + 1, current_cell[1])
                elif open_direction == 'N':
                    child_cell = (current_cell[0] - 1, current_cell[1])

                if child_cell in explored_cells:
                    continue

                explored_cells.append(child_cell)
                cells_to_explore.append(child_cell)
                explored_paths[child_cell] = current_cell

    bfs_path = {}
    cell = (1, 1)
    while cell != (maze.rows, maze.cols):
        bfs_path[explored_paths[cell]] = cell
        cell = explored_paths[cell]
    return bfs_path, explored_cells


if __name__ == '__main__':
    m = maze(5,5)
    m.CreateMaze(loopPercent=50)
    start_time = time.time()
    bfs_path, explored_cells_path = bfs(m)
    end_time = time.time()
    total_time_taken = end_time - start_time
    print(total_time_taken)
    no_explored = len(explored_cells_path)
    total_cells = m.rows * m.cols
    cells_in_bfs_path = len(bfs_path)
    print(no_explored)
    print(total_cells)
    print(cells_in_bfs_path)

    Label1 = textLabel(m, "Total time taken", total_time_taken)
    Label2 = textLabel(m, "Total cells explored", no_explored)
    Label3 = textLabel(m, "Shortest path", cells_in_bfs_path)
    a = agent(m, footprints=True, filled=False)
    b = agent(m, footprints=True, color=COLOR.green)
    m.tracePath({b: explored_cells_path}, delay=100)
    m.tracePath({a: bfs_path}, delay=100)

    #m.run()
```

DepthFirstSearch.py

```
from pyamaze import maze, agent, COLOR
import time


def dfs(maze):
    explored_cells = []
    cells_to_explore = [(maze.rows, maze.cols)]
    explored_paths = {}

    #For first iteration
    explored_cells.append((maze.rows, maze.cols))

    while len(cells_to_explore) > 0:
        current_cell = cells_to_explore.pop()

        # Goal reached condition
        if current_cell == (1, 1):
            break

        for open_direction in 'ESNW':
            if maze.maze_map[current_cell][open_direction]:
                if open_direction == 'E':
                    child_cell = (current_cell[0], current_cell[1] + 1)
                elif open_direction == 'W':
                    child_cell = (current_cell[0], current_cell[1] - 1)
                elif open_direction == 'S':
                    child_cell = (current_cell[0] + 1, current_cell[1])
                elif open_direction == 'N':
                    child_cell = (current_cell[0] - 1, current_cell[1])

                if child_cell in explored_cells:
                    continue

                explored_cells.append(child_cell)
                cells_to_explore.append(child_cell)
                explored_paths[child_cell] = current_cell

    dfs_path = {}
    cell = (1, 1)
    while cell != (maze.rows, maze.cols):
        dfs_path[explored_paths[cell]] = cell
        cell = explored_paths[cell]
    return dfs_path, explored_cells


if __name__ == '__main__':
    m = maze(5,5)
    m.CreateMaze(loopPercent=50)
    start_time = time.time()
    dfs_path, explored_cells_path = dfs(m)
    end_time = time.time()
    total_time_taken = end_time-start_time
    print(total_time_taken)
    no_explored = len(explored_cells_path)
    total_cells = m.rows*m.cols
    cells_in_dfs_path = len(dfs_path)
    print(no_explored)
    print(total_cells)
    print(cells_in_dfs_path)
    a = agent(m, footprints=True, filled=False)
    b = agent(m, footprints=True, color=COLOR.green)
    m.tracePath({b: explored_cells_path}, delay=100)
    m.tracePath({a: dfs_path}, delay=100)

    #m.run()
```

A_Star.py

```
from pyamaze import maze, agent, COLOR, textLabel
import time
from queue import PriorityQueue


def manhattan_distance(cell1, cell2):
    x1, y1 = cell1
    x2, y2 = cell2
    return (abs(x1 - x2) + abs(y1 - y2))
```

```
# heuristic
def h_score(cell1, cell2):
    return manhattan_distance(cell1, cell2)


def a_star(maze):
    start_cell = (maze.rows, maze.cols)
    goal_cell = (1,1)

    g_score = {cell: float('inf') for cell in maze.grid}
    f_score = {cell: float('inf') for cell in maze.grid}

    g_score[start_cell] = 0
    # f = g + h
    f_score[start_cell] = g_score[start_cell] +
h_score(start_cell,goal_cell)

    open_cells = PriorityQueue()

    open_cells.put((h_score(start_cell, goal_cell), h_score(start_cell,
goal_cell), start_cell))

    explored_path ={}
    explored_cells = []

    while not open_cells.empty():
        current_cell = open_cells.get()[2]

        if current_cell not in explored_cells:
            explored_cells.append(current_cell)

        if current_cell == goal_cell:
            break

        for open_direction in 'ESNW':
            if maze.maze_map[current_cell][open_direction]:
                if open_direction == 'E':
                    child_cell = (current_cell[0], current_cell[1] + 1)
                elif open_direction == 'W':
                    child_cell = (current_cell[0], current_cell[1] - 1)
                elif open_direction == 'S':
                    child_cell = (current_cell[0] + 1, current_cell[1])
                elif open_direction == 'N':
                    child_cell = (current_cell[0] - 1, current_cell[1])

                updated_g_score = g_score[current_cell]+1
                updated_f_score = updated_g_score +
h_score(child_cell,goal_cell)

                if updated_f_score < f_score[child_cell]:
                    g_score[child_cell] = updated_g_score
                    f_score[child_cell] = updated_f_score
                    open_cells.put((f_score[child_cell],
h_score(child_cell, goal_cell), child_cell))
                    explored_path[child_cell] = current_cell

    a_star_path = {}
    cell = goal_cell
    while cell!=start_cell:
        a_star_path[explored_path[cell]] = cell
        cell = explored_path[cell]

    return a_star_path,explored_cells

if __name__ == '__main__':
    m = maze(5, 5)
    m.CreateMaze(loopPercent=100)

    a_star_path, explored_path = a_star(m)

    a = agent(m, footprints=True, filled=False)
    b = agent(m, footprints=True, color=COLOR.green)
    m.tracePath({b: explored_path}, delay=100)
    m.tracePath({a: a_star_path}, delay=100)

    #m.run()
```

PolicyIteration.py

```
from pyamaze import maze, agent, textLabel
import numpy as np
import time


def initialisation(maze):
    maze.states = {}
    maze.states = list(maze.maze_map.keys())

    maze.rewards = {}
    for state in maze.states:
        if state == (1, 1):
            maze.rewards[state] = 1
        else:
            maze.rewards[state] = -0.04

    maze.valueMap = {}
    for state in maze.states:
        maze.valueMap[state] = 0

    maze.transitions = {}
    for state in maze.states:
        maze.transitions[state] = {}
        for action in possibleActions(maze, state):
            nextState = getNextState(maze, state, action)
            if nextState == state:
                maze.transitions[state][action] = [(1, nextState)]
            else:
                maze.transitions[state][action] = [(0.9, nextState),
(0.1, state)]

    maze.policy = {}
    for state in maze.states:
        maze.policy[state] = {np.random.choice(possibleActions(maze,
state)): 1}

    maze.gamma = 0.99
    maze.theta = 1e-3


def possibleActions(maze, state):
    (x, y) = state
    actions = maze.maze_map[(x, y)]
    possibleStates = []
    for direction, bool in actions.items():
        if bool == 1:
            possibleStates.append(direction)
    return possibleStates


def getNextState(maze, currentState, action):
    if action == 'E':
        nextState = (currentState[0], currentState[1] + 1)
    elif action == 'W':
        nextState = (currentState[0], currentState[1] - 1)
    elif action == 'S':
        nextState = (currentState[0] + 1, currentState[1])
    elif action == 'N':
        nextState = (currentState[0] - 1, currentState[1])
    else:
        nextState = currentState
    if nextState in maze.states:
        return nextState
    else:
        return currentState


def policyEvaluation(maze):
    # print("Eval start")
    while True:
        delta = 0
        for state in maze.states:
            v = 0
            for action, actionProb in maze.policy[state].items():
                for prob, nextState in maze.transitions[state][action]:
                    v += actionProb * prob * (maze.rewards[nextState] +
maze.gamma * maze.valueMap[nextState])
            delta = max(delta, abs(v - maze.valueMap[state]))
            maze.valueMap[state] = v
        if delta < maze.theta:
            break
    # print("Eval done")


def policyImprovement(maze):
    policyStable = True
    for state in maze.states:
        oldAction = max(maze.policy[state], key=maze.policy[state].get)
        newAction = None
        maxValue = float('-inf')
        for action in possibleActions(maze, state):
            v = 0
            for prob, nextState in maze.transitions[state][action]:
                v += prob * (maze.rewards[nextState] + maze.gamma *
maze.valueMap[nextState])
```

```
            if v > maxValue:
                maxValue = v
                newAction = action
        maze.policy[state] = {newAction: 1}
        if oldAction != newAction:
            policyStable = False
    return policyStable


def getPath(maze):
    # print("Path start")
    path = []
    start = (maze.rows, maze.cols)
    current = start
    goal = (1, 1)
    path.append(current)
    while current != goal:
        action = max(maze.policy[current],
key=maze.policy[current].get)
        current = getNextState(maze, current, action)
        path.append(current)
    # print("Path done")
    return path


def policyIteration(maze):
    initialisation(maze)
    while True:
        policyEvaluation(maze)
        if policyImprovement(maze):
            break
    return getPath(maze)


if __name__ == '__main__':
    m = maze(5, 5)
    m.CreateMaze()
    start_time = time.time()
    policyIterationPath = policyIteration(m)
    end_time = time.time()
    # print(policyIterationPath)

    total_time_taken = end_time-start_time
    print(total_time_taken)

    cells_in_policy_iteration_path = len(policyIterationPath)
    print(cells_in_policy_iteration_path)

    Label1 = textLabel(m,"Total time taken", total_time_taken)
    Label3 = textLabel(m, "Shortest path",
cells_in_policy_iteration_path)

    a = agent(m, footprints=True, filled=False)
    m.tracePath({a: policyIterationPath}, delay=100)

    m.run()
```

ValueIteration.py

```
import numpy as np
from pyamaze import maze, agent, COLOR, textLabel
import time


def initialisation(maze):
    maze.states = {}
    maze.states = list(maze.maze_map.keys())

    maze.rewards = {}
    for state in maze.states:
        if state == (1, 1):
            maze.rewards[state] = 1
        else:
            maze.rewards[state] = -0.04

    maze.policy = {}
    for state in maze.states:
        maze.policy[state] = np.random.choice(possibleActions(maze,
state))

    maze.valueMap = {}
    for state in maze.states:
        if state == (1, 1):
            maze.valueMap[state] = 10000
        else:
            maze.valueMap[state] = -1

    maze.threshold = 0.005
    maze.discount = 0.9


def possibleActions(maze, state):
    (x, y) = state
    actions = maze.maze_map[(x, y)]
    possibleStates = []
    for direction, bool in actions.items():
        if bool == 1:
            possibleStates.append(direction)
    return possibleStates


def getNextState(maze, currentState, action):
    if action == 'E':
        nextState = (currentState[0], currentState[1] + 1)
    elif action == 'W':
        nextState = (currentState[0], currentState[1] - 1)
    elif action == 'S':
        nextState = (currentState[0] + 1, currentState[1])
    elif action == 'N':
        nextState = (currentState[0] - 1, currentState[1])
    else:
        nextState = currentState
    if nextState in maze.states:
        return nextState
    else:
        return currentState
```

```
def valueIteration(maze):

    initialisation(maze)

    while True:
        delta = 0
        for state in maze.states:
            oldValue = maze.valueMap[state]
            max_value = float('-inf')
            for action in possibleActions(maze, state):
                nextState = getNextState(maze, state, action)
                v = maze.rewards[state] + (maze.discount *
maze.valueMap[nextState])
                if v > max_value:
                    max_value = v
                    maze.policy[state] = action
            maze.valueMap[state] = max_value
            delta = max(delta, abs(oldValue - maze.valueMap[state]))
            #print(delta)
        if delta < maze.threshold:
            break

    return getPath(maze)


def getPath(maze):
    # print("Path start")
    path = []
    start = (maze.rows, maze.cols)
    current = start
    goal = (1, 1)
    path.append(current)
    while current != goal:
        action = maze.policy[current]
        current = getNextState(maze, current, action)
        path.append(current)
    # print("Path done")
    return path


if __name__ == '__main__':
    m = maze(5, 5)
    m.CreateMaze()
    start_time = time.time()
    valueIterationPath = valueIteration(m)
    end_time = time.time()
    # print(policyIterationPath)

    total_time_taken = end_time-start_time
    print(total_time_taken)

    cells_in_value_iteration_path = len(valueIterationPath)
    print(cells_in_value_iteration_path)

    Label1 = textLabel(m,"Total time taken", total_time_taken)
    Label3 = textLabel(m, "Shortest path",
cells_in_value_iteration_path)
```

```
    a = agent(m, footprints=True, filled=False)
    m.tracePath({a: valueIterationPath}, delay=100)

    m.run()
```

Comparer1.py

```
import os
import psutil
from MazeGenerator import generate_maze
from BreadthFirstSearch import bfs
from DepthFirstSearch import dfs
from A_Star import a_star
import time
import pandas as pd
import gc


def memory_usage():
    process = psutil.Process(os.getpid())
    mem_info = process.memory_info()
    return mem_info.rss / (1024 ** 2)  # Return memory usage in MB


maze_dimensions = [5, 10, 20, 50, 100]
loop_percent = [0, 50, 100]
patterns = ["none", 'v', 'h']
results = []

no_of_iterations = 10


for dimensions in maze_dimensions:
    bfs_metrics = [0, 0, 0, 0]  # path length, cells explored, time
taken, memory used
    dfs_metrics = [0, 0, 0, 0]
    a_star_metrics = [0, 0, 0, 0]
    for iteration in range(no_of_iterations):
        maze = generate_maze(dimensions, dimensions, 50, "none")

        mem_before = memory_usage()
        bfs_start_time = time.perf_counter()
        bfs_path, bfs_explored_cells_path = bfs(maze)
        bfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        bfs_metrics[0] += len(bfs_path)+1
        bfs_metrics[1] += len(bfs_explored_cells_path)
        bfs_metrics[2] += (bfs_end_time - bfs_start_time)
        bfs_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        dfs_start_time = time.perf_counter()
        dfs_path, dfs_explored_cells_path = dfs(maze)
        dfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        dfs_metrics[0] += len(dfs_path)+1
        dfs_metrics[1] += len(dfs_explored_cells_path)
        dfs_metrics[2] += (dfs_end_time - dfs_start_time)
        dfs_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        a_star_start_time = time.perf_counter()
        a_star_path, a_star_explored_cells_path = a_star(maze)
        a_star_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        a_star_metrics[0] += len(a_star_path)+1
        a_star_metrics[1] += len(a_star_explored_cells_path)
        a_star_metrics[2] += (a_star_end_time - a_star_start_time)
        a_star_metrics[3] += (mem_after - mem_before)

    # Calculate averages
    bfs_avg = [x / no_of_iterations for x in bfs_metrics]
    dfs_avg = [x / no_of_iterations for x in dfs_metrics]
    a_star_avg = [x / no_of_iterations for x in a_star_metrics]

    # Append the results for each algorithm
    results.append(['BFS', dimensions] + bfs_avg)
    results.append(['DFS', dimensions] + dfs_avg)
    results.append(['A*', dimensions] + a_star_avg)

# Create a DataFrame
df = pd.DataFrame(results,
                  columns=['Algorithm', 'Dimensions', 'Shortest Path
Length', 'No of Cells Explored', 'Time Taken (s)',
                           'Memory Usage (MB)'])

df.to_csv("Compare1_Dimensions.csv", index=False)

# Display the table
print(df)

results = []

for loop in loop_percent:
    bfs_metrics = [0, 0, 0, 0]  # path length, cells explored, time
taken, memory used
    dfs_metrics = [0, 0, 0, 0]
    a_star_metrics = [0, 0, 0, 0]

    for iteration in range(no_of_iterations):
        maze = generate_maze(100, 100, loop, "none")

        mem_before = memory_usage()
        bfs_start_time = time.perf_counter()
        bfs_path, bfs_explored_cells_path = bfs(maze)
        bfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        bfs_metrics[0] += len(bfs_path)+1
        bfs_metrics[1] += len(bfs_explored_cells_path)
        bfs_metrics[2] += (bfs_end_time - bfs_start_time)
        bfs_metrics[3] += (mem_after - mem_before)
        mem_before = memory_usage()
        dfs_start_time = time.perf_counter()
        dfs_path, dfs_explored_cells_path = dfs(maze)
        dfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        dfs_metrics[0] += len(dfs_path)+1
        dfs_metrics[1] += len(dfs_explored_cells_path)
        dfs_metrics[2] += (dfs_end_time - dfs_start_time)
        dfs_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        a_star_start_time = time.perf_counter()
        a_star_path, a_star_explored_cells_path = a_star(maze)
        a_star_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        a_star_metrics[0] += len(a_star_path)+1
        a_star_metrics[1] += len(a_star_explored_cells_path)
        a_star_metrics[2] += (a_star_end_time - a_star_start_time)
        a_star_metrics[3] += (mem_after - mem_before)


    # Calculate averages
    bfs_avg = [x / no_of_iterations for x in bfs_metrics]
    dfs_avg = [x / no_of_iterations for x in dfs_metrics]
    a_star_avg = [x / no_of_iterations for x in a_star_metrics]

    # Append the results for each algorithm
    results.append(['BFS', loop] + bfs_avg)
    results.append(['DFS', loop] + dfs_avg)
    results.append(['A*', loop] + a_star_avg)

# Create a DataFrame
df = pd.DataFrame(results,
                  columns=['Algorithm', 'Loop Percent(%)', 'Shortest
Path Length', 'No of Cells Explored', 'Time Taken (s)',
                           'Memory Usage (MB)'])

df.to_csv("Compare1_loops.csv", index=False)

# Display the table
print(df)

results = []


for pattern in patterns:
    bfs_metrics = [0, 0, 0, 0]  # path length, cells explored, time
taken, memory used
    dfs_metrics = [0, 0, 0, 0]
    a_star_metrics = [0, 0, 0, 0]

    for iteration in range(no_of_iterations):
        maze = generate_maze(100, 100, 50, pattern)

        mem_before = memory_usage()
        bfs_start_time = time.perf_counter()
        bfs_path, bfs_explored_cells_path = bfs(maze)
        bfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        bfs_metrics[0] += len(bfs_path)+1
        bfs_metrics[1] += len(bfs_explored_cells_path)
        bfs_metrics[2] += (bfs_end_time - bfs_start_time)
        bfs_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        dfs_start_time = time.perf_counter()
        dfs_path, dfs_explored_cells_path = dfs(maze)
        dfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        dfs_metrics[0] += len(dfs_path)+1
        dfs_metrics[1] += len(dfs_explored_cells_path)
        dfs_metrics[2] += (dfs_end_time - dfs_start_time)
        dfs_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        a_star_start_time = time.perf_counter()
        a_star_path, a_star_explored_cells_path = a_star(maze)
        a_star_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        a_star_metrics[0] += len(a_star_path)+1
        a_star_metrics[1] += len(a_star_explored_cells_path)
        a_star_metrics[2] += (a_star_end_time - a_star_start_time)
        a_star_metrics[3] += (mem_after - mem_before)

    # Calculate averages
    bfs_avg = [x / no_of_iterations for x in bfs_metrics]
    dfs_avg = [x / no_of_iterations for x in dfs_metrics]
    a_star_avg = [x / no_of_iterations for x in a_star_metrics]

    # Append the results for each algorithm
    results.append(['BFS', pattern] + bfs_avg)
    results.append(['DFS', pattern] + dfs_avg)
    results.append(['A*', pattern] + a_star_avg)

# Create a DataFrame
df = pd.DataFrame(results,
                  columns=['Algorithm', 'Pattern', 'Shortest Path
Length', 'No of Cells Explored', 'Time Taken (s)',
                           'Memory Usage (MB)'])

df.to_csv("Compare1_pattern.csv", index=False)

# Display the table
print(df)
```

Comparer2.py

```
import os
import psutil
from MazeGenerator import generate_maze
from PolicyIteration import policyIteration
from ValueIteration import valueIteration
```

```
import time
import pandas as pd
import gc

def memory_usage():
    process = psutil.Process(os.getpid())
    mem_info = process.memory_info()
    return mem_info.rss / (1024 ** 2)  # Return memory usage in MB


maze_dimensions = [5, 10, 20, 50, 100]
loop_percent = [100,50,0]
patterns = ["none", 'v', 'h']
results = []

no_of_iterations = 10

for dimensions in maze_dimensions:
    PI_metrics = [0, 0, 0]
    VI_metrics = [0, 0, 0]

    for iteration in range(no_of_iterations):
        maze = generate_maze(dimensions, dimensions, 50, "none")

        mem_before = memory_usage()
        PI_start_time = time.perf_counter()
        PI_path = policyIteration(maze)
        PI_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        PI_metrics[0] += len(PI_path)
        PI_metrics[1] += (PI_end_time - PI_start_time)
        PI_metrics[2] += mem_after - mem_before

        mem_before = memory_usage()
        VI_start_time = time.perf_counter()
        VI_path = valueIteration(maze)
        VI_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        VI_metrics[0] += len(VI_path)
        VI_metrics[1] += (VI_end_time - VI_start_time)
        VI_metrics[2] += mem_after - mem_before

    PI_avg = [x / no_of_iterations for x in PI_metrics]
    VI_avg = [x / no_of_iterations for x in VI_metrics]

    results.append(['Policy Iteration', dimensions] + PI_avg)
    results.append(['Value Iteration', dimensions] + VI_avg)

df = pd.DataFrame(results,
                  columns=['Algorithm', 'Dimensions', 'Shortest Path
Length', 'Time Taken (s)', 'Memory Usage (MB)'])

df.to_csv("Compare2_Dimensions.csv", index=False)

print(df)
```

```
results = []

for loop in loop_percent:
    PI_metrics = [0, 0, 0]
    VI_metrics = [0, 0, 0]

    for iteration in range(no_of_iterations):
        maze = generate_maze(20, 20, loop, "none")

        mem_before = memory_usage()
        PI_start_time = time.perf_counter()
        PI_path = policyIteration(maze)
        PI_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        PI_metrics[0] += len(PI_path)
        PI_metrics[1] += (PI_end_time - PI_start_time)
        PI_metrics[2] += mem_after - mem_before

        if loop == 0:
            continue
        else:
            mem_before = memory_usage()
            VI_start_time = time.perf_counter()
            VI_path = valueIteration(maze)
            VI_end_time = time.perf_counter()
            gc.collect()
            mem_after = memory_usage()
            VI_metrics[0] += len(VI_path)
            VI_metrics[1] += (VI_end_time - VI_start_time)
            VI_metrics[2] += mem_after - mem_before

    PI_avg = [x / no_of_iterations for x in PI_metrics]
    VI_avg = [x / no_of_iterations for x in VI_metrics]

    results.append(['Policy Iteration', loop] + PI_avg)
    results.append(['Value Iteration', loop] + VI_avg)

df = pd.DataFrame(results,
                  columns=['Algorithm', 'Loop Percent (%)', 'Shortest
Path Length', 'Time Taken (s)', 'Memory Usage (MB)'])

df.to_csv("Compare2_loops.csv", index=False)

print(df)

results = []

for pattern in patterns:
    PI_metrics = [0, 0, 0]
    VI_metrics = [0, 0, 0]

    for iteration in range(no_of_iterations):
        maze = generate_maze(20, 20, 50, pattern)

        mem_before = memory_usage()
        PI_start_time = time.perf_counter()
        PI_path = policyIteration(maze)
        PI_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        PI_metrics[0] += len(PI_path)
        PI_metrics[1] += (PI_end_time - PI_start_time)
        PI_metrics[2] += mem_after - mem_before

        mem_before = memory_usage()
        VI_start_time = time.perf_counter()
        VI_path = valueIteration(maze)
        VI_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        VI_metrics[0] += len(VI_path)
        VI_metrics[1] += (VI_end_time - VI_start_time)
        VI_metrics[2] += mem_after - mem_before

    PI_avg = [x / no_of_iterations for x in PI_metrics]
    VI_avg = [x / no_of_iterations for x in VI_metrics]

    results.append(['Policy Iteration', pattern] + PI_avg)
    results.append(['Value Iteration', pattern] + VI_avg)

df = pd.DataFrame(results,
                  columns=['Algorithm', 'Pattern', 'Shortest Path
Length', 'Time Taken (s)', 'Memory Usage (MB)'])

df.to_csv("Compare2_pattern.csv", index=False)

print(df)
```

Comparer3.py

```
import os
import psutil
from MazeGenerator import generate_maze
from BreadthFirstSearch import bfs
from DepthFirstSearch import dfs
from A_Star import a_star
from PolicyIteration import policyIteration
from ValueIteration import valueIteration
import time
import pandas as pd
import gc


def memory_usage():
    process = psutil.Process(os.getpid())
    mem_info = process.memory_info()
    return mem_info.rss / (1024 ** 2)  # Return memory usage in MB


maze_dimensions = [5, 10, 20, 50, 100]
loop_percent = [0, 50, 100]
patterns = ["none", 'v', 'h']
results = []

no_of_iterations = 10
```

```
#
# for dimensions in maze_dimensions:
#     bfs_metrics = [0, 0, 0, 0]  # path length, cells explored, time
taken, memory used
#     dfs_metrics = [0, 0, 0, 0]
#     a_star_metrics = [0, 0, 0, 0]
#     PI_metrics = [0, 0, 0, 0]
#     VI_metrics = [0, 0, 0, 0]
#     for iteration in range(no_of_iterations):
#         maze = generate_maze(dimensions, dimensions, 50, "none")
#
#         mem_before = memory_usage()
#         bfs_start_time = time.perf_counter()
#         bfs_path, bfs_explored_cells_path = bfs(maze)
#         bfs_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         bfs_metrics[0] += len(bfs_path)+1
#         bfs_metrics[1] += len(bfs_explored_cells_path)
#         bfs_metrics[2] += (bfs_end_time - bfs_start_time)
#         bfs_metrics[3] += (mem_after - mem_before)
#
#         mem_before = memory_usage()
#         dfs_start_time = time.perf_counter()
#         dfs_path, dfs_explored_cells_path = dfs(maze)
#         dfs_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         dfs_metrics[0] += len(dfs_path)+1
#         dfs_metrics[1] += len(dfs_explored_cells_path)
#         dfs_metrics[2] += (dfs_end_time - dfs_start_time)
#         dfs_metrics[3] += (mem_after - mem_before)
#
#         mem_before = memory_usage()
#         a_star_start_time = time.perf_counter()
#         a_star_path, a_star_explored_cells_path = a_star(maze)
#         a_star_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         a_star_metrics[0] += len(a_star_path)+1
#         a_star_metrics[1] += len(a_star_explored_cells_path)
#         a_star_metrics[2] += (a_star_end_time - a_star_start_time)
#         a_star_metrics[3] += (mem_after - mem_before)
#
#         mem_before = memory_usage()
#         PI_start_time = time.perf_counter()
#         PI_path = policyIteration(maze)
#         PI_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         PI_metrics[0] += len(PI_path)
#         PI_metrics[2] += (PI_end_time - PI_start_time)
#         PI_metrics[3] += mem_after - mem_before
#
#         mem_before = memory_usage()
#         VI_start_time = time.perf_counter()
#         VI_path = valueIteration(maze)
#         VI_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         VI_metrics[0] += len(VI_path)
#         VI_metrics[2] += (VI_end_time - VI_start_time)
#         VI_metrics[3] += mem_after - mem_before
#
#     # Calculate averages
#     bfs_avg = [x / no_of_iterations for x in bfs_metrics]
#     dfs_avg = [x / no_of_iterations for x in dfs_metrics]
#     a_star_avg = [x / no_of_iterations for x in a_star_metrics]
#     PI_avg = [x / no_of_iterations for x in PI_metrics]
#     VI_avg = [x / no_of_iterations for x in VI_metrics]
#
#     # Append the results for each algorithm
#     results.append(['BFS', dimensions] + bfs_avg)
#     results.append(['DFS', dimensions] + dfs_avg)
#     results.append(['A*', dimensions] + a_star_avg)
#     results.append(['Policy Iteration', dimensions] + PI_avg)
#     results.append(['Value Iteration', dimensions] + VI_avg)
#
# # Create a DataFrame
# df = pd.DataFrame(results,
#                   columns=['Algorithm', 'Dimensions', 'Shortest Path
Length', 'No of Cells Explored', 'Time Taken (s)',
#                            'Memory Usage (MB)'])
#
# df.to_csv("Compare3_Dimensions.csv", index=False)
#
# # Display the table
# print(df)

# results = []
#
# for loop in loop_percent:
#     bfs_metrics = [0, 0, 0, 0]  # path length, cells explored, time
taken, memory used
#     dfs_metrics = [0, 0, 0, 0]
#     a_star_metrics = [0, 0, 0, 0]
#     PI_metrics = [0, 0, 0, 0]
#     VI_metrics = [0, 0, 0, 0]
#
#     for iteration in range(no_of_iterations):
#         print(iteration)
#         maze = generate_maze(20, 20, loop, "none")
#
#         mem_before = memory_usage()
#         bfs_start_time = time.perf_counter()
#         bfs_path, bfs_explored_cells_path = bfs(maze)
#         bfs_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         bfs_metrics[0] += len(bfs_path)+1
#         bfs_metrics[1] += len(bfs_explored_cells_path)
#         bfs_metrics[2] += (bfs_end_time - bfs_start_time)
#         bfs_metrics[3] += (mem_after - mem_before)
#         print("bfs")
#
#         mem_before = memory_usage()
#         dfs_start_time = time.perf_counter()
#         dfs_path, dfs_explored_cells_path = dfs(maze)
#         dfs_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         dfs_metrics[0] += len(dfs_path)+1
#         dfs_metrics[1] += len(dfs_explored_cells_path)
#         dfs_metrics[2] += (dfs_end_time - dfs_start_time)
#         dfs_metrics[3] += (mem_after - mem_before)
#         print("dfs")
#
#         mem_before = memory_usage()
#         a_star_start_time = time.perf_counter()
#         a_star_path, a_star_explored_cells_path = a_star(maze)
#         a_star_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         a_star_metrics[0] += len(a_star_path)+1
#         a_star_metrics[1] += len(a_star_explored_cells_path)
#         a_star_metrics[2] += (a_star_end_time - a_star_start_time)
#         a_star_metrics[3] += (mem_after - mem_before)
#         print("a*")
#
#         mem_before = memory_usage()
#         PI_start_time = time.perf_counter()
#         PI_path = policyIteration(maze)
#         PI_end_time = time.perf_counter()
#         gc.collect()
#         mem_after = memory_usage()
#         PI_metrics[0] += len(PI_path)
#         PI_metrics[2] += (PI_end_time - PI_start_time)
#         PI_metrics[3] += mem_after - mem_before
#         print("PI")
#
#         if loop == 0:
#             continue
#         else:
#             mem_before = memory_usage()
#             VI_start_time = time.perf_counter()
#             VI_path = valueIteration(maze)
#             VI_end_time = time.perf_counter()
#             gc.collect()
#             mem_after = memory_usage()
#             VI_metrics[0] += len(VI_path)
#             VI_metrics[2] += (VI_end_time - VI_start_time)
#             VI_metrics[3] += mem_after - mem_before
#             print("VI")
#
#
#     # Calculate averages
#     bfs_avg = [x / no_of_iterations for x in bfs_metrics]
#     dfs_avg = [x / no_of_iterations for x in dfs_metrics]
#     a_star_avg = [x / no_of_iterations for x in a_star_metrics]
#     PI_avg = [x / no_of_iterations for x in PI_metrics]
#     VI_avg = [x / no_of_iterations for x in VI_metrics]
#
#     # Append the results for each algorithm
#     results.append(['BFS', loop] + bfs_avg)
#     results.append(['DFS', loop] + dfs_avg)
#     results.append(['A*', loop] + a_star_avg)
#     results.append(['Policy Iteration', loop] + PI_avg)
#     results.append(['Value Iteration', loop] + VI_avg)
#
# # Create a DataFrame
# df = pd.DataFrame(results,
#                   columns=['Algorithm', 'Loop Percent(%)', 'Shortest
Path Length', 'No of Cells Explored', 'Time Taken (s)',
#                            'Memory Usage (MB)'])
#
# df.to_csv("Compare3_loops.csv", index=False)
#
# # Display the table
# print(df)
#
# results = []


for pattern in patterns:
    bfs_metrics = [0, 0, 0, 0]  # path length, cells explored, time
taken, memory used
    dfs_metrics = [0, 0, 0, 0]
    a_star_metrics = [0, 0, 0, 0]
    PI_metrics = [0, 0, 0, 0]
    VI_metrics = [0, 0, 0, 0]

    for iteration in range(no_of_iterations):
        maze = generate_maze(20, 20, 50, pattern)

        mem_before = memory_usage()
        bfs_start_time = time.perf_counter()
        bfs_path, bfs_explored_cells_path = bfs(maze)
        bfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        bfs_metrics[0] += len(bfs_path)+1
        bfs_metrics[1] += len(bfs_explored_cells_path)
        bfs_metrics[2] += (bfs_end_time - bfs_start_time)
        bfs_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        dfs_start_time = time.perf_counter()
        dfs_path, dfs_explored_cells_path = dfs(maze)
        dfs_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        dfs_metrics[0] += len(dfs_path)+1
        dfs_metrics[1] += len(dfs_explored_cells_path)
        dfs_metrics[2] += (dfs_end_time - dfs_start_time)
        dfs_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        a_star_start_time = time.perf_counter()
        a_star_path, a_star_explored_cells_path = a_star(maze)
        a_star_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        a_star_metrics[0] += len(a_star_path)+1
        a_star_metrics[1] += len(a_star_explored_cells_path)
        a_star_metrics[2] += (a_star_end_time - a_star_start_time)
        a_star_metrics[3] += (mem_after - mem_before)

        mem_before = memory_usage()
        PI_start_time = time.perf_counter()
        PI_path = policyIteration(maze)
        PI_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        PI_metrics[0] += len(PI_path)
        PI_metrics[2] += (PI_end_time - PI_start_time)
        PI_metrics[3] += mem_after - mem_before

        mem_before = memory_usage()
        VI_start_time = time.perf_counter()
        VI_path = valueIteration(maze)
        VI_end_time = time.perf_counter()
        gc.collect()
        mem_after = memory_usage()
        VI_metrics[0] += len(VI_path)
        VI_metrics[2] += (VI_end_time - VI_start_time)
        VI_metrics[3] += mem_after - mem_before

    # Calculate averages
    bfs_avg = [x / no_of_iterations for x in bfs_metrics]
    dfs_avg = [x / no_of_iterations for x in dfs_metrics]
    a_star_avg = [x / no_of_iterations for x in a_star_metrics]
    PI_avg = [x / no_of_iterations for x in PI_metrics]
    VI_avg = [x / no_of_iterations for x in VI_metrics]

    # Append the results for each algorithm
    results.append(['BFS', pattern] + bfs_avg)
    results.append(['DFS', pattern] + dfs_avg)
    results.append(['A*', pattern] + a_star_avg)
    results.append(['Policy Iteration', pattern] + PI_avg)
    results.append(['Value Iteration', pattern] + VI_avg)

# Create a DataFrame
df = pd.DataFrame(results,
                  columns=['Algorithm', 'Pattern', 'Shortest Path
Length', 'No of Cells Explored', 'Time Taken (s)',
                           'Memory Usage (MB)'])

df.to_csv("Compare3_pattern.csv", index=False)

# Display the table
print(df)
```