Name: Ketan Patil
Student Id: 22303876

Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE

Design Choices

1. Maze Generator Module

I have used **Pyamaze** module to generate mazes for this assignment. It is an open-source module with free to use License. I have attached license documentation in the Appendix section of this report. The code for this library is available on Github. There are two ways to use this library:

a) Download the Pyamaze.py code from GitHub repository and include it in the source code directory for using the API functions. I have provided GitHub repository link in the Appendix section of this report.
b) Install the library using '**pip'** command and use it directly afterwards. I have provided running instructions for this in Running Instructions section of this report

I have used the second approach because it is easier to implement. The primary reason for using this library is that it provides very simple functions to generate maze of any size. Also, it provides an option to save mazes into CSV format. This is a significant feature which can be used to compare performance of different algorithms on same set of mazes.

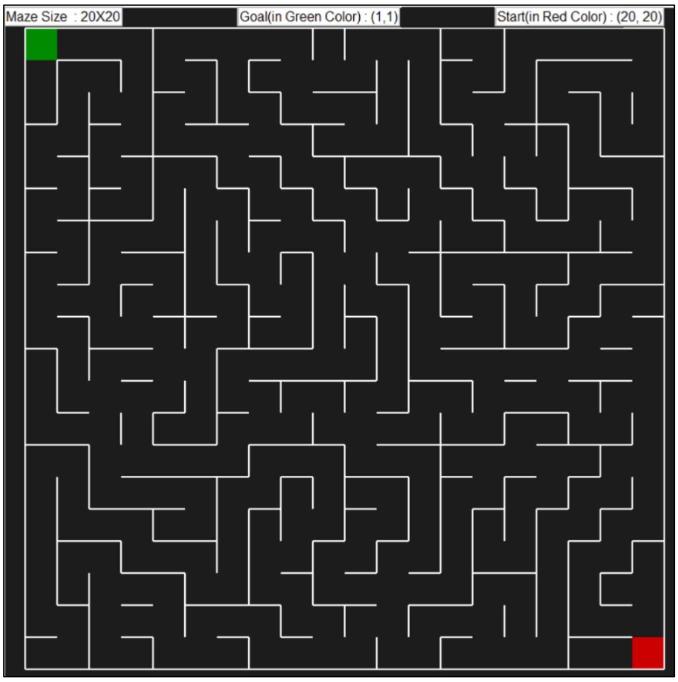
Furthermore, this API also provides simple functions to generate visualisations for mazes and provide features to track the path of algorithms as it traverses the maze.

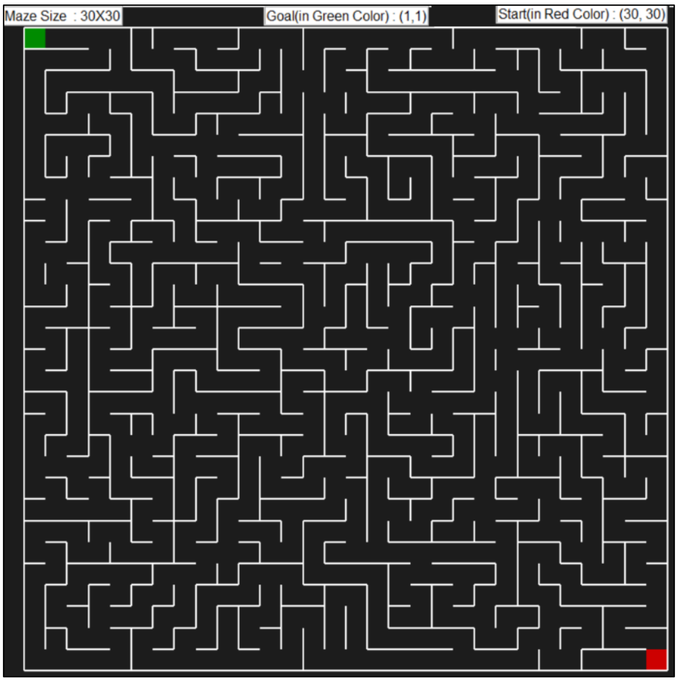
2. Maze Size

Choosing the right gris is very important factor while analysing the performance of various algorithms. The primary reason is that execution of algorithms on appropriate maze size can help us to gauge the performance of algorithm on various parameters. This can help us to understand the difference in behaviour of different algorithms under distinct scenarios and enable us to choose right algorithm for specific types of problems. I have tried multiple maze sizes for each of these algorithms. I have tried following maze sizes: 5X5, 7X7, 10X10, 15X15, 20X20, 25X25, 30X30, 40X40, and 80X80.

For maze of size less than 20X20, I was not able to see comprehensive difference in the performance parameters primarily time taken by algorithm to finish execution and memory consumption of the algorithm. For maze of size greater than 40X40 some algorithms failed to converge.

Therefore, I have chosen to evaluate these algorithms on 3 maze sizes: 20X20, 30X30, and 40X40. Below is the visual representation of these mazes:





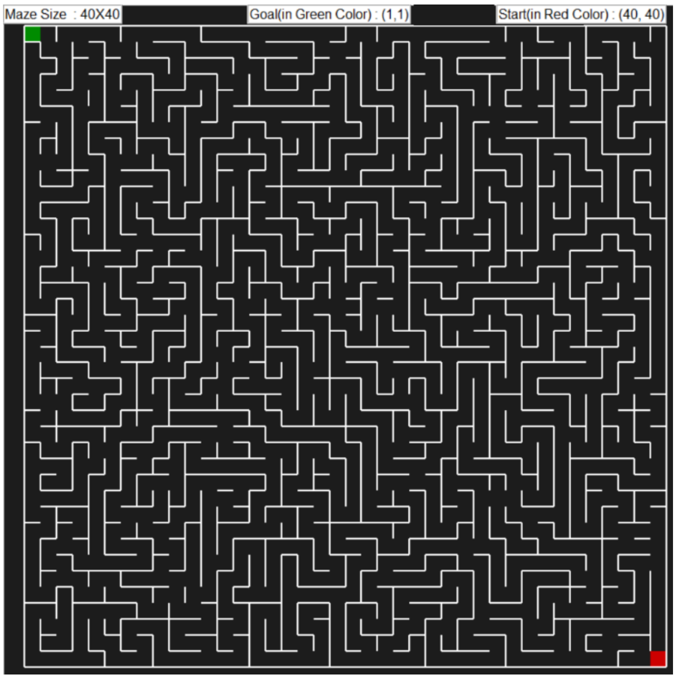


Figure 1: Maze of size 20X20, 30X30, and 40X40

3. Heuristic for A* algorithm

A* algorithm uses different heuristics to identify the optimal path to goal instead of exploring all possible node while searching for the goal. Heuristic is a way to estimate the distance between the current node and goal node.

The primary reason for evaluating performance of A* algorithm on different heuristics is that heuristic plays a significant role to optimise the performance of A* algorithm and ensures that it find the optimal path to goal efficiently by not exploring the paths that are too costly or will not lead to goal.

I have tried two different heuristics:

a) Manhattan Distance
b) Euclidean Distance

4. Gamma or Discount Factor

Discount factor plays a very important role in achieving convergence in MDP algorithms. It motivates the search algorithm to search further for the goal. As rewards get discounted by each step taken towards the goal, therefore this factor can play significant factor in finding the goal node in maze.

I have analysed MDP algorithms for multiple discount factors like 0.1, 0.5, and 0.9. For all the 3 maze sizes that I have selected, I was able to find goal for discount factor of 0.9.

Name: Ketan Patil
Student Id: 22303876

Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE

Comparisons of different algorithms

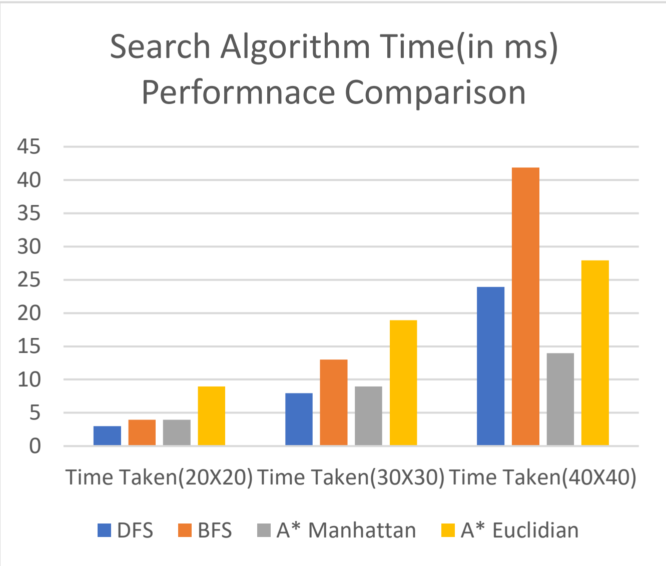
I have compared performance these algorithms on different parameters. The primary reason for this is that it will help us the gauge their performance under different scenarios and help us better understand their functioning.

5. Time taken to execute code

Below table summarises the time taken in milliseconds to execute each of these algorithms for different maze sizes:

Maze Size	DFS	BFS	A* Manhattan	A* Euclidian	MDP Value Iteration	MDP Policy Iteration
Time Taken(20X20)	2.99	3.98	3.98	8.97	136.66	194.47
Time Taken(30X30)	7.97	13	8.97	18.94	331.14	565.48
Time Taken(40X40)	23.93	41.88	13.96	27.92	606.37	1389.27

Table 1: Performance of different algorithms on metric: Time



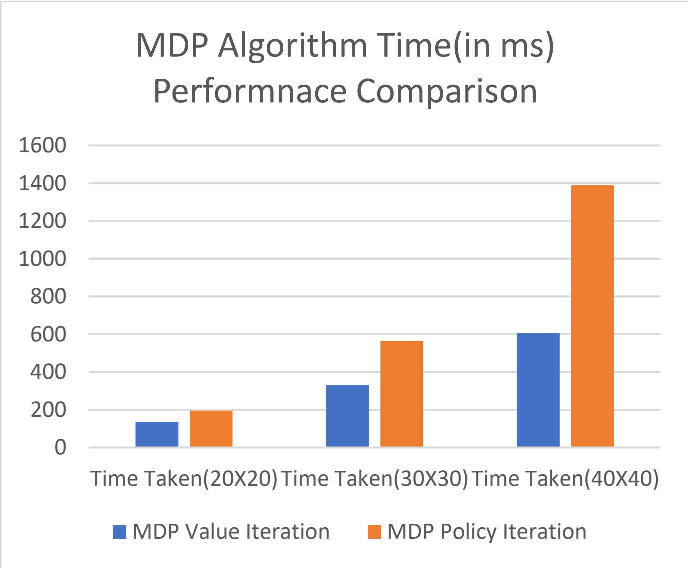


Figure 2: Graphs comparing performance of different algorithms on metric: Time

Please note that I have created 2 different charts for Search and MDP algorithms as the scale of time these algorithms was very different and plotting them on same graph was making it difficult to visualise.

From above graphs I can conclude that for search algorithms, DFS and A* algorithms work very efficiently with respect to time of execution and are able to find goal in shortest possible time if the maze size is small. However, for bigger maze sizes like 40X40, A* with Manhattan distance as heuristic outperforms all algorithms.

For MDP algorithms, Value Iteration works better than Policy Iteration.

When I compare the performance of Search Algorithms with MDP algorithms, Search algorithms perform significantly better because these involve a smaller number of evaluation and calculations in each iteration as compared to MDP algorithms.

Furthermore, A* algorithm with Manhattan distance as heuristic outperforms Euclidean distance for all maze sizes.

6. Number of steps in shortest path to Goal

Below table summarises the number of steps in the shortest path to goal for each of these algorithms for different maze sizes:

Maze Size	DFS	BFS	A* Manhattan	A* Euclidian	MDP Value Iteration	MDP Policy Iteration
Shortest Path to goal(20X20)	75	65	65	65	65	65
Shortest Path to goal(30X30)	273	99	99	99	99	99
Shortest Path to goal(40X40)	305	127	127	127	127	127

Table 2: Performance of different algorithms on metric: Shortest path to goal

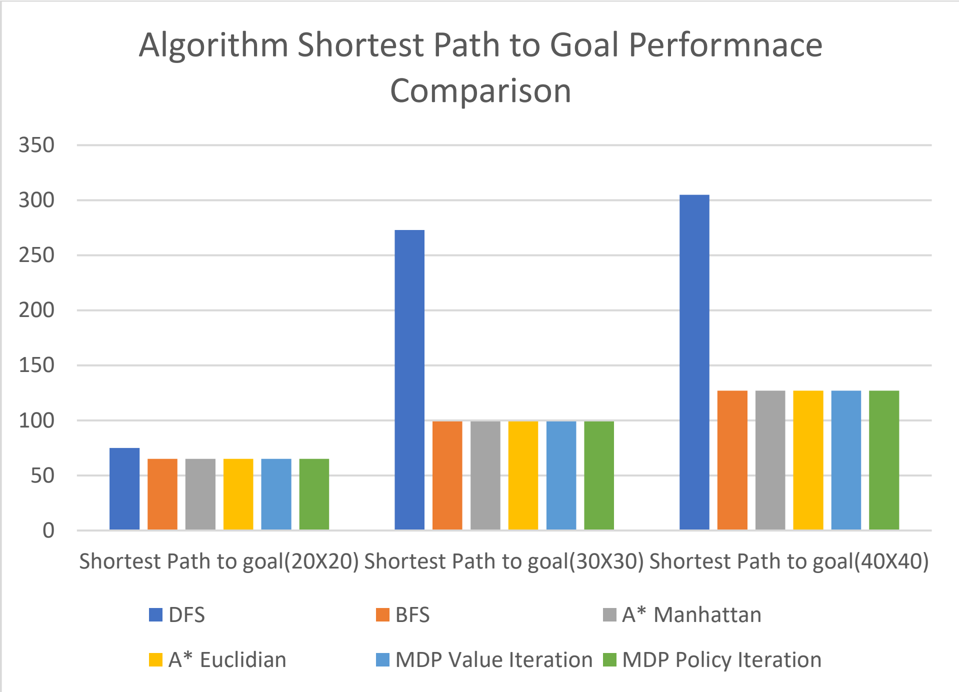


Figure 3: Graphs comparing performance of different algorithms on metric: shortest path to goal

From above graph I can conclude that all algorithms except DFS were able to identify shortest path to goal in same number of steps irrespective of the maze size.

DFS took maximum steps to goal for all maze sizes.

7. Nodes explored to reach goal

Below table summarises the number of nodes explored by each of these algorithms to reach the goal for different maze sizes:

Maze Size	DFS	BFS	A* Manhattan	A* Euclidian	MDP Value Iteration	MDP Policy Iteration
Number of Nodes Explored(20X20)	395	396	330	359	All Nodes	All Nodes
Number of Nodes Explored(30X30)	650	812	713	772	All Nodes	All Nodes
Number of Nodes Explored(40X40)	1154	1457	1091	1210	All Nodes	All Nodes

Table 3: Performance of different algorithms on metric: Nodes explored to reach goal

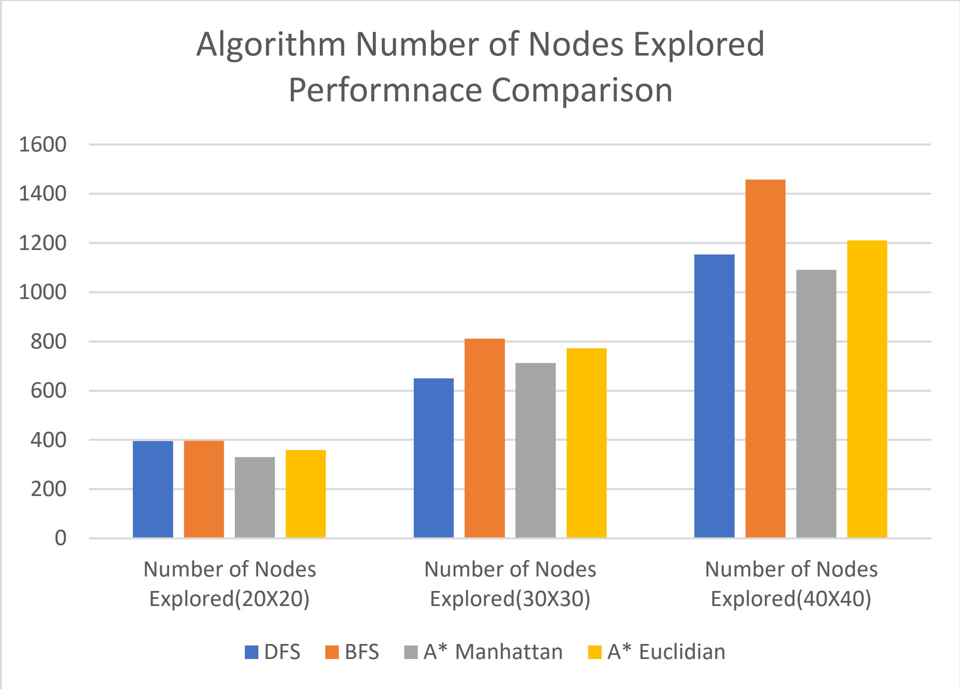


Figure 4: Graphs comparing performance of different algorithms on metric: shortest path to goal

From above graph I can conclude that A* algorithm explores minimum nodes in order to reach goal for all maze sizes. Primary reason for this behaviour is that this algorithm uses heuristics to identify the optimal path to goal unlike blind search performed by DFS and BFS algorithms.

Furthermore, BFS explores maximum number of nodes while searching for the goal. However, MDP algorithms have to explore all nodes in order to evaluate the optimal path to goal.

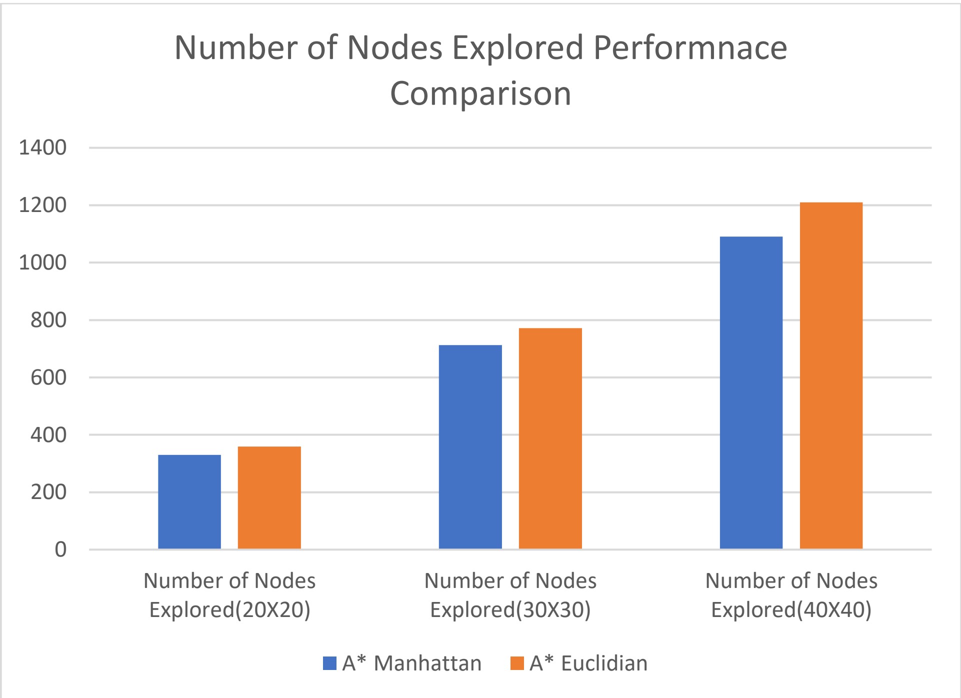


Figure 5: Graphs comparing performance of different heuristic for A* algorithm

For A* algorithm, Manhattan distance outperforms Euclidean distance for all maze sizes.

8. Memory Consumed

Below table summarises the memory consumed in MB to execute each of these algorithms for different maze sizes:

Maze Size	DFS	BFS	A* Manhattan	A* Euclidian	MDP Value Iteration	MDP Policy Iteration
Memory Consumed(20X20)	0.032	0.032	0.016	0.039	0.029	0.028
Memory Consumed(30X30)	0.032	0.063	0.062	0.077	0.064	0.046
Memory Consumed(40X40)	0.064	0.128	0.062	0.077	0.112	0.085

Table 4: Performance of different algorithms on metric: Memory Consumed

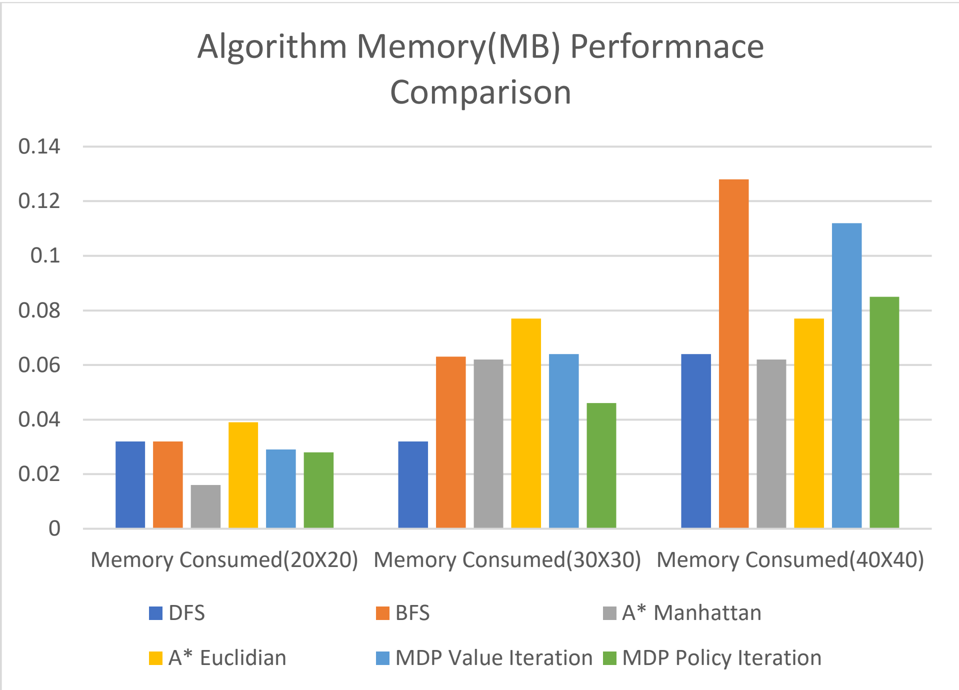


Figure 6: Graphs comparing performance of different algorithms on metric: Memory

From above graphs I can conclude that for search algorithms DFS and A* algorithms work very efficiently with respect to memory consumed for execution. Also, for bigger maze size BFS consumes significantly more memory than all other algorithms.

For MDP algorithms, Value iteration consumes more memory than Policy iteration

Furthermore, A* algorithm with Manhattan distance as heuristic outperforms Euclidean distance for all maze sizes.

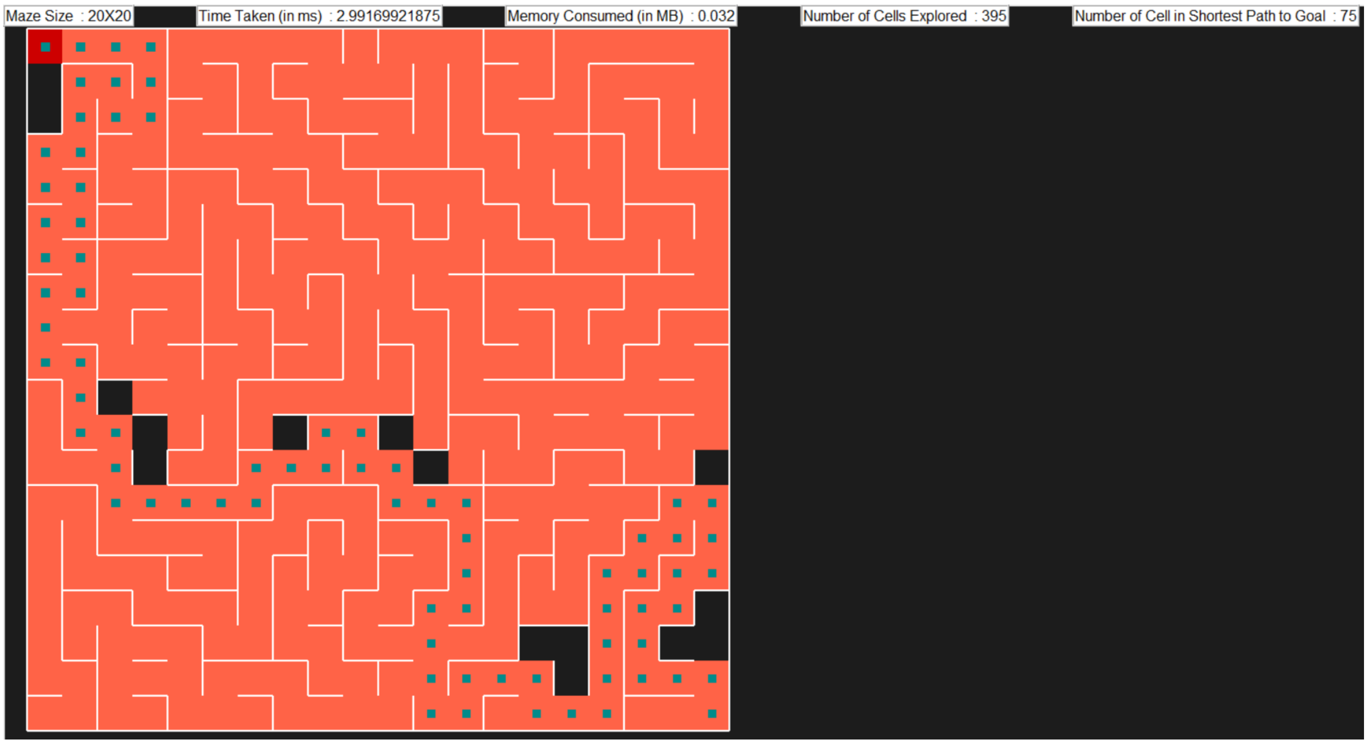
Conclusion

I can conclude that behaviour of these algorithms varies with maze size therefore, we should analyse the complexity of the problem before applying these algorithms. Performance of Search and MDP based algorithms are very different due to the number of evaluations involved in each step. MDP algorithms will always provide optimal policy to reach goal but they take more time and explore entire maze to evaluate the optimal policy. For Search based algorithms, informed search algorithm like A* will generally perform better than blind search algorithms like BFS and DFS. Furthermore, Manhattan distance is the ideal heuristic for A* algorithm.

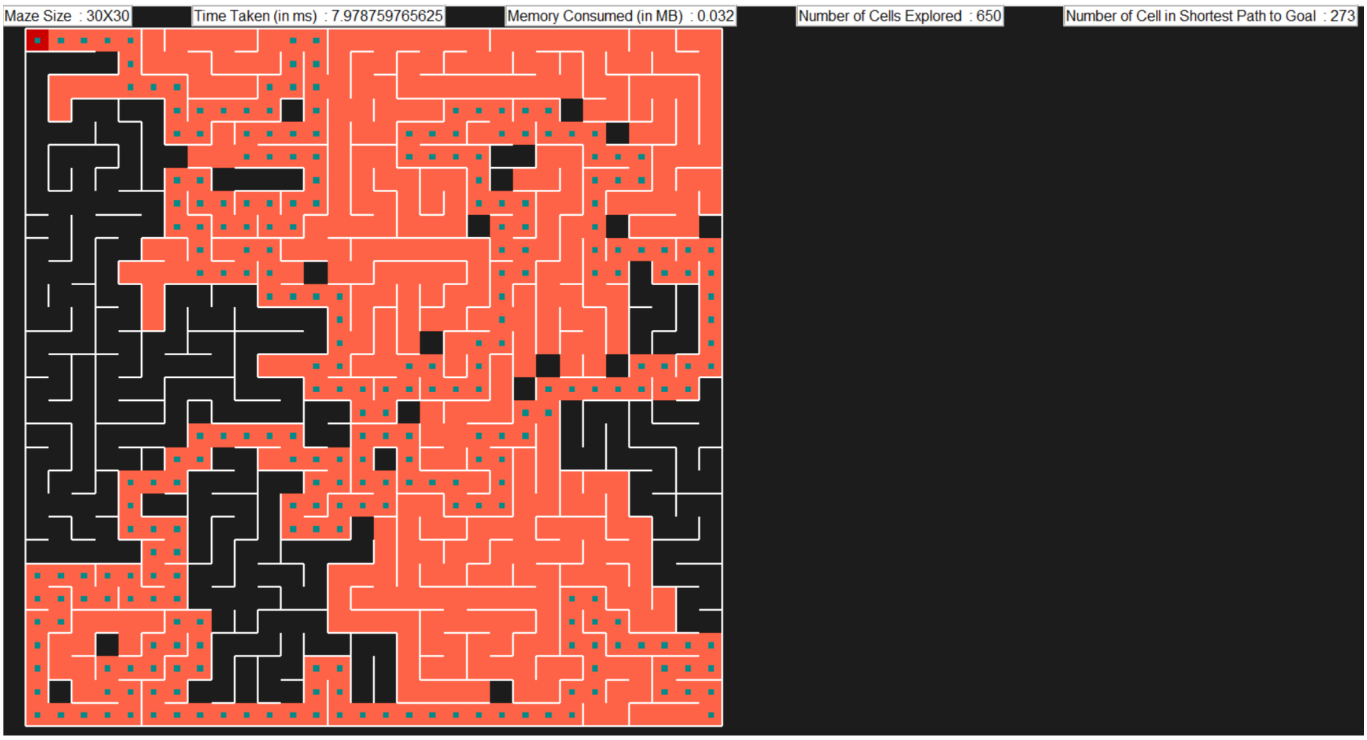
Visualisation of the Performance of all algorithms

1. Depth First Search

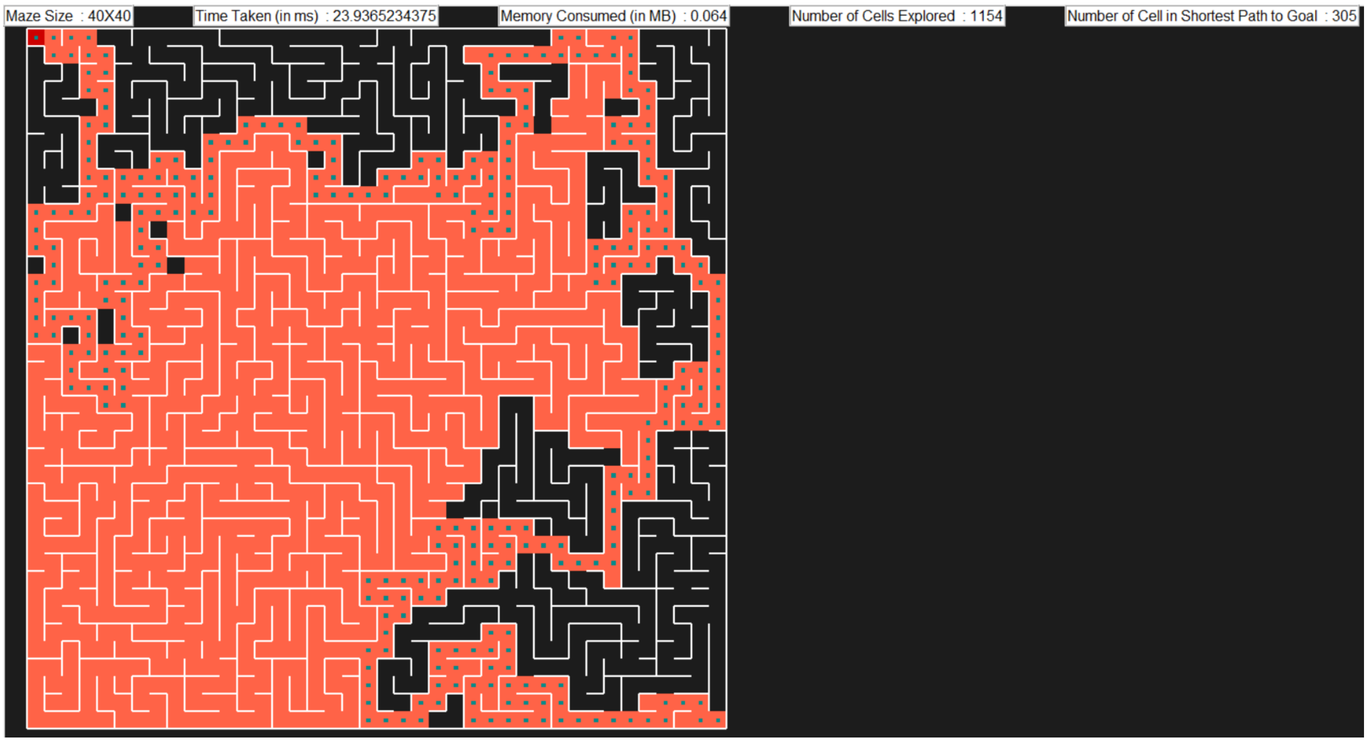
	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	20X20	2.991699	0.032000	395	75



	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	30X30	7.978760	0.032000	650	273

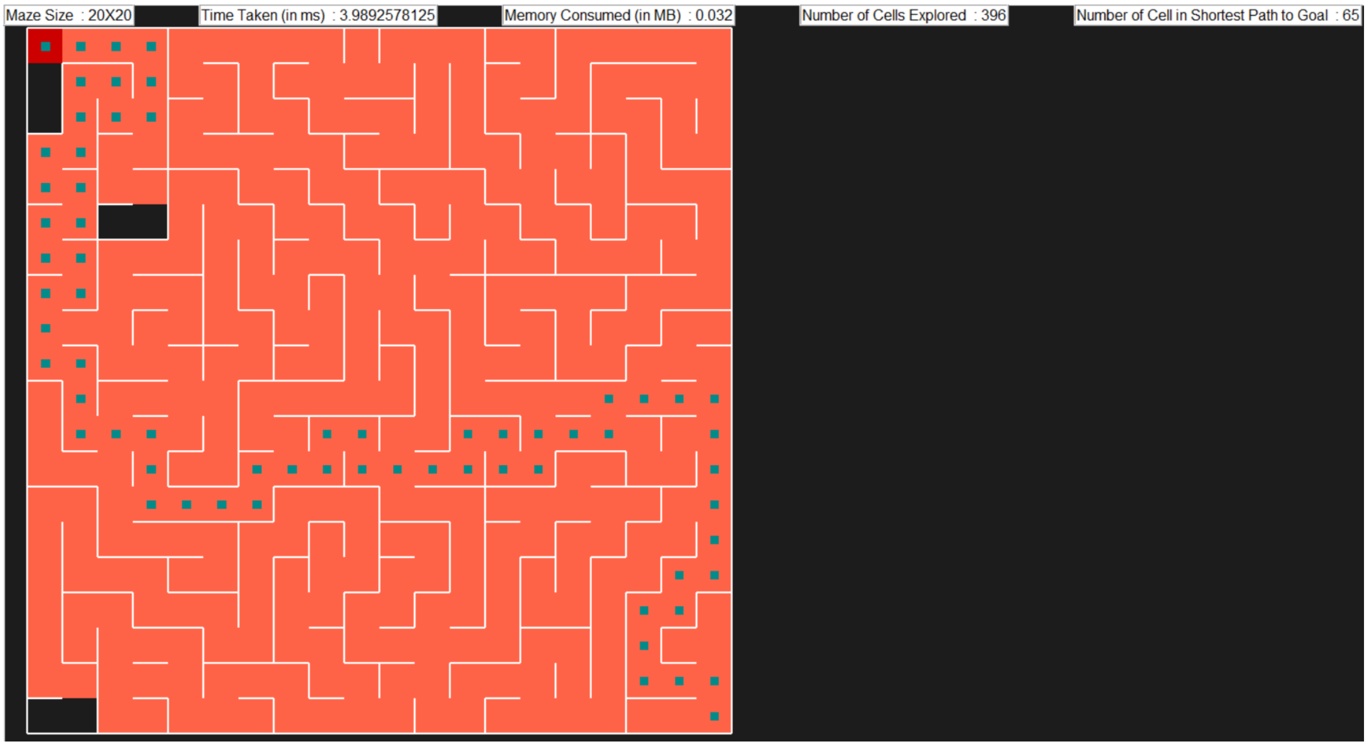


	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	40X40	23.936523	0.064000	1154	305



2. Breadth First Search

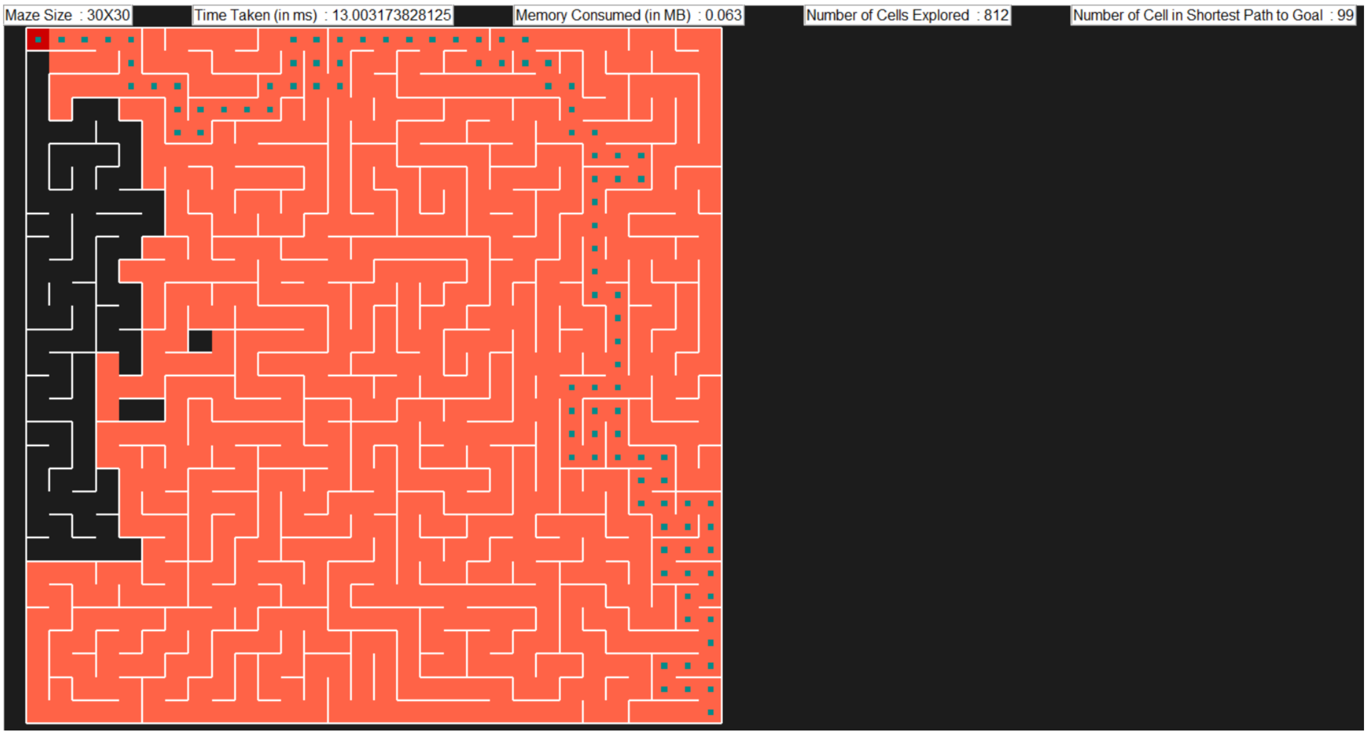
	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	20X20	3.989258	0.032000	396	65



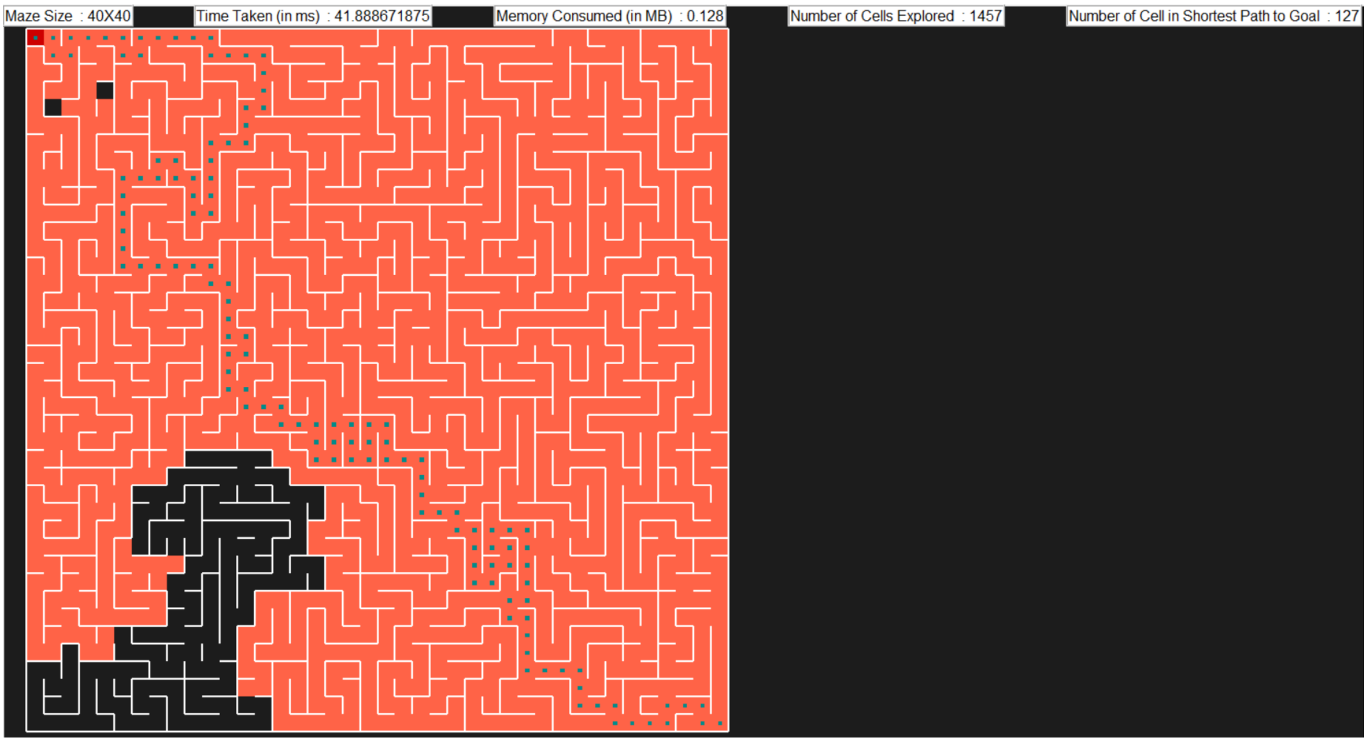
Name: Ketan Patil
Student Id: 22303876

Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	30X30	13.003174	0.063000	812	99



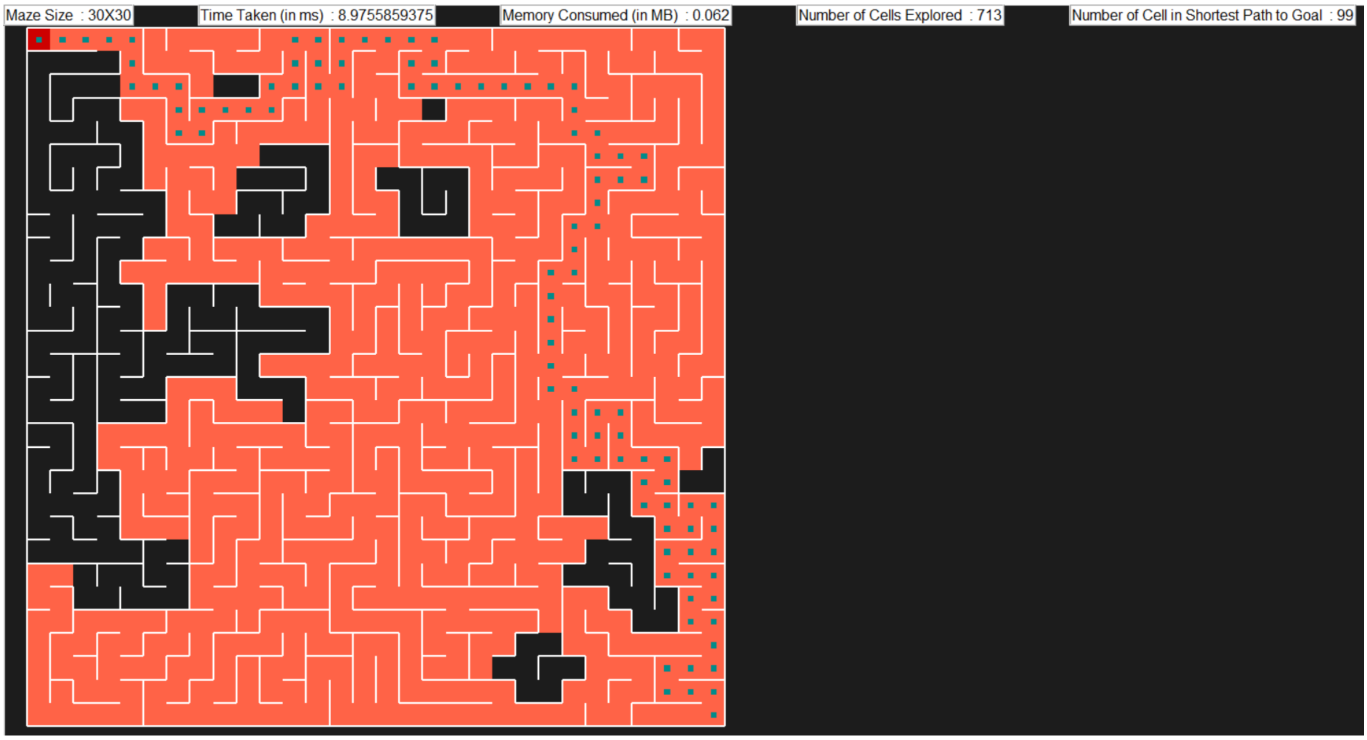
	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	40X40	41.888672	0.128000	1457	127



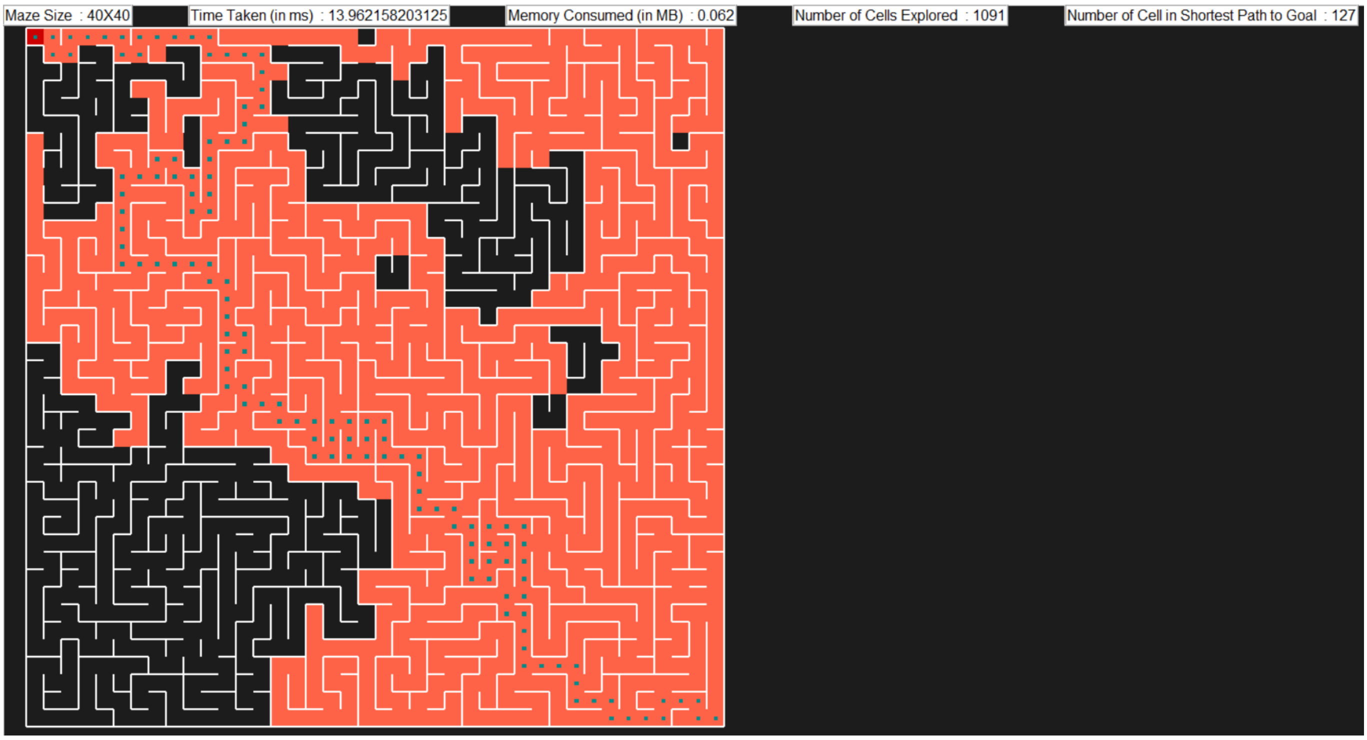
3. A Star Search with Manhattan Distance as Heuristics

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	20X20	3.988525	0.016000	330	65

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	30X30	8.975586	0.062000	713	99



	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	40X40	13.962158	0.062000	1091	127



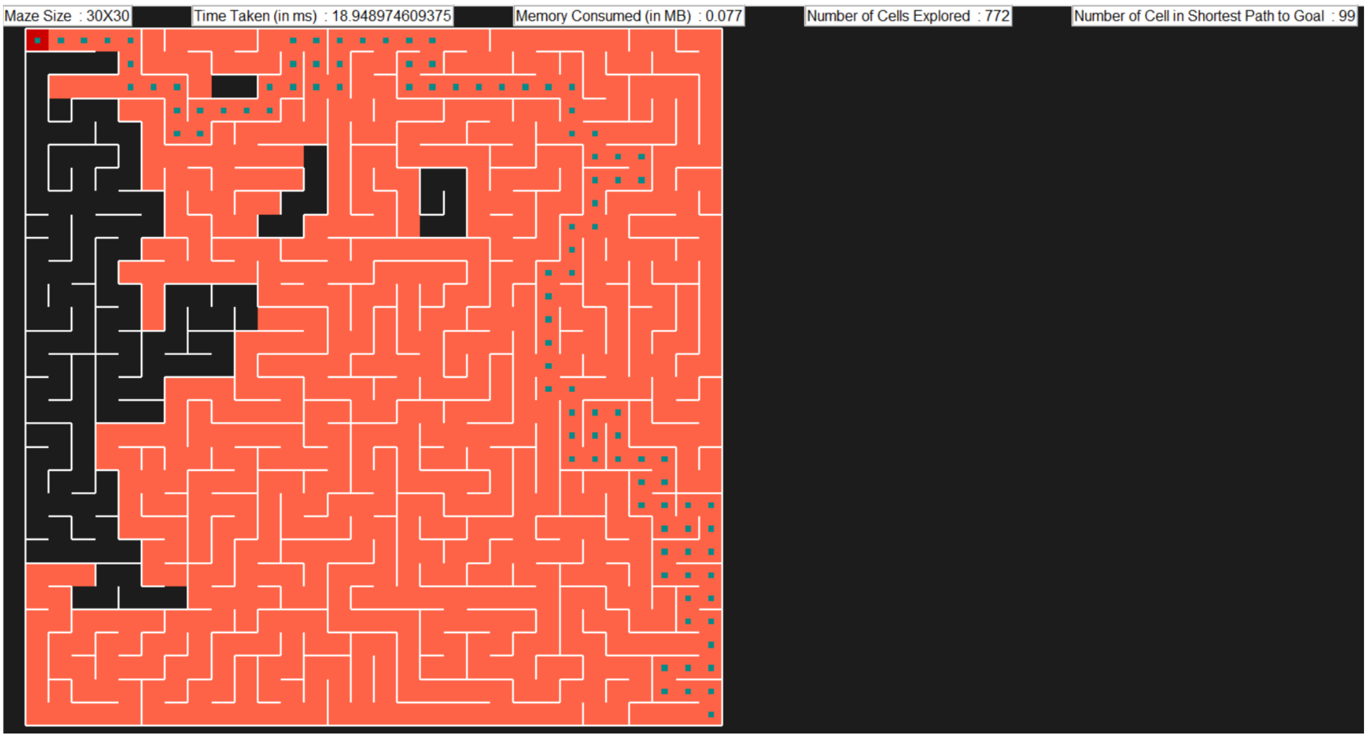
4. A Star Search with Euclidian Distance as Heuristics

Name: Ketan Patil
Student Id: 22303876

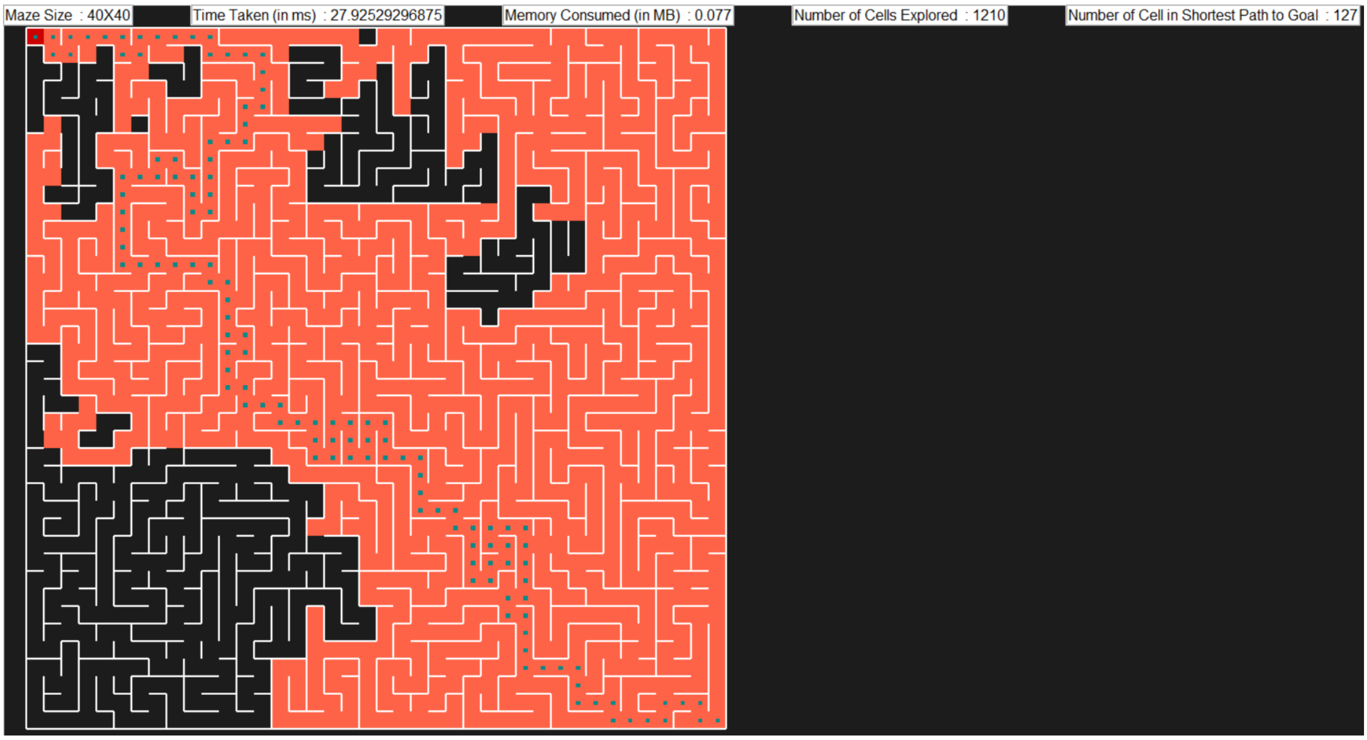
Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	20X20	8.975830	0.039000	359	65

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	30X30	18.948975	0.077000	772	99



	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cells Explored	Number of Cell in Shortest Path to Goal
0	40X40	27.925293	0.077000	1210	127



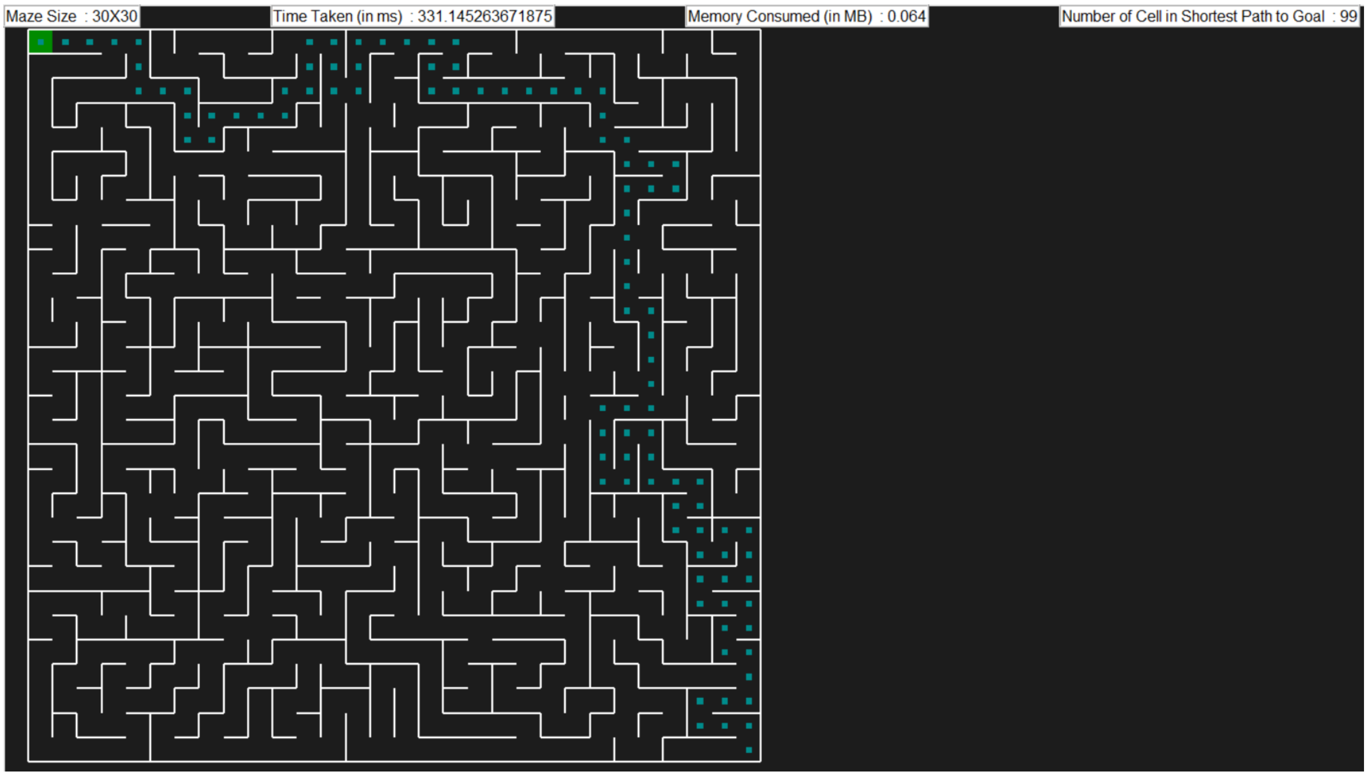
5. MDP Value Iteration

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cell in Shortest Path to Goal
0	20X20	136.666992	0.029000	65

Name: Ketan Patil
Student Id: 22303876

Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE

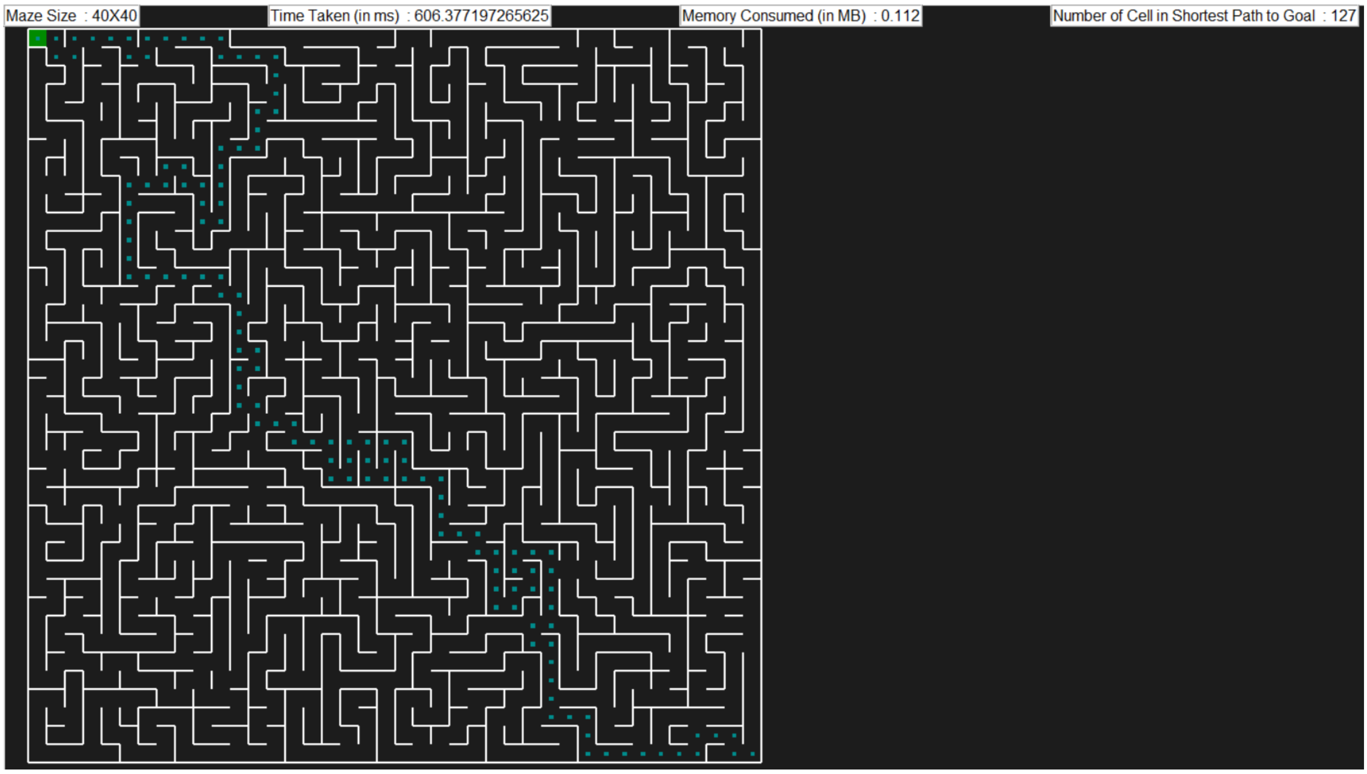
	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cell in Shortest Path to Goal
0	30X30	331.145264	0.064000	99



	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cell in Shortest Path to Goal
0	40X40	606.377197	0.112000	127

Name: Ketan Patil
Student Id: 22303876

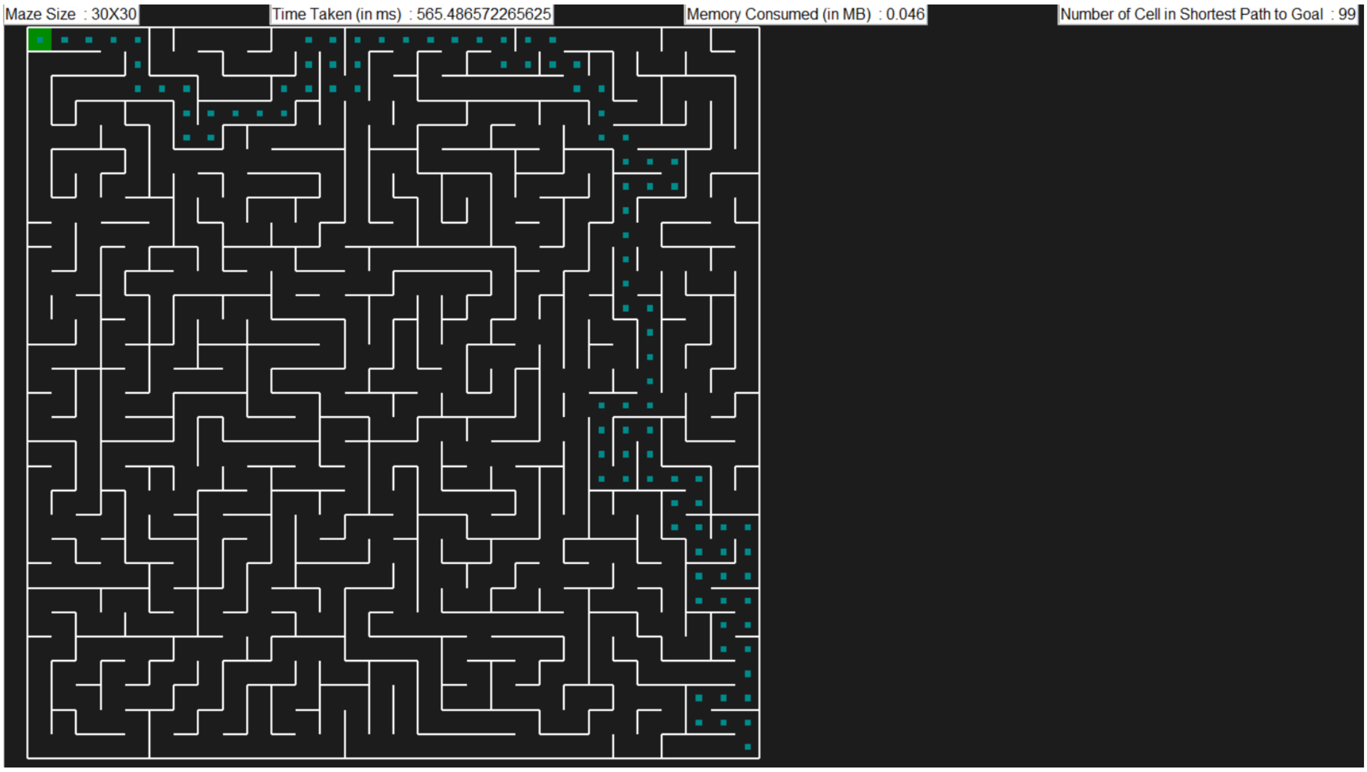
Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE



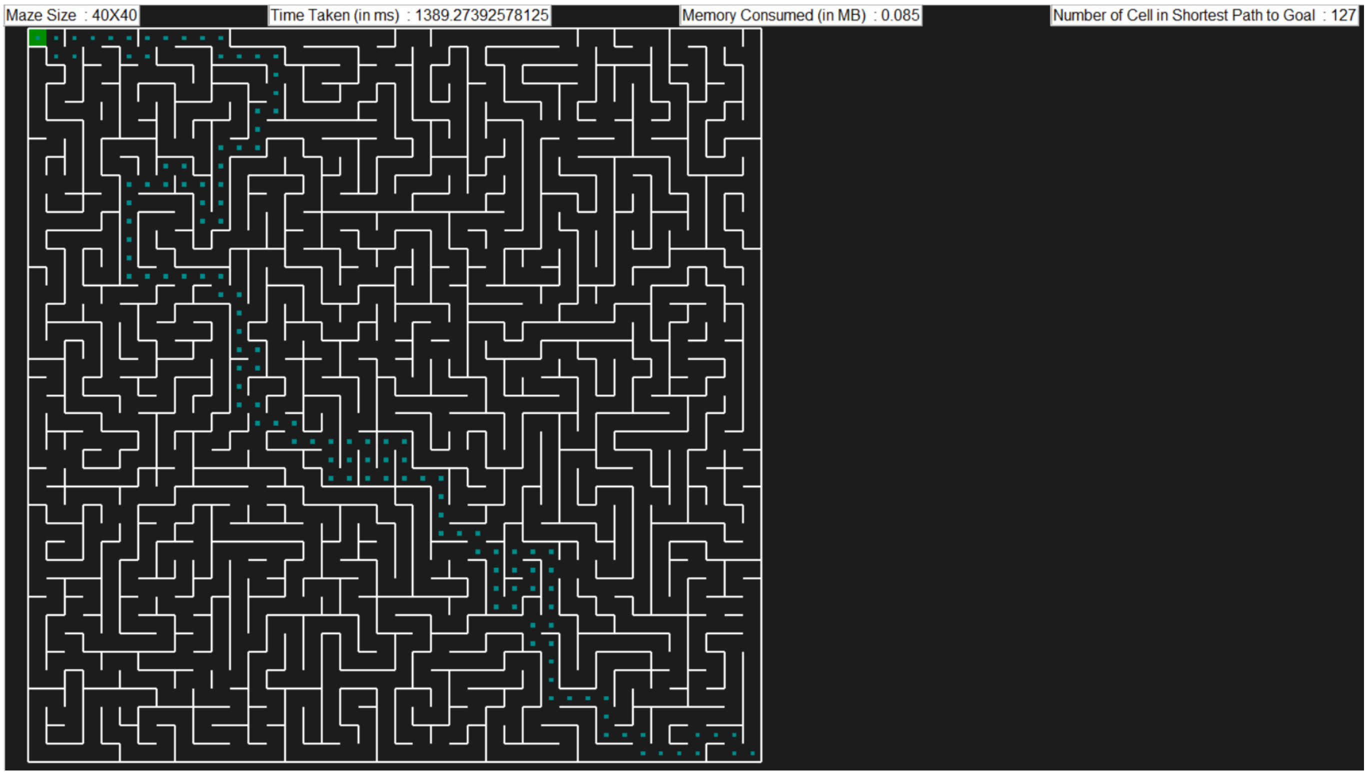
6. MDP Policy Iteration

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cell in Shortest Path to Goal
0	20X20	194.478516	0.028000	65

	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cell in Shortest Path to Goal
0	30X30	565.486572	0.046000	99



	Maze Size	Time Taken (in ms)	Memory Consumed (in MB)	Number of Cell in Shortest Path to Goal
0	40X40	1389.273926	0.085000	127



References

i. https://github.com/MAN1986/pyamaze/tree/main/Demos

ii. https://github.com/SparkShen02/MDP-with-Value-Iteration-and-Policy-Iteration
iii. https://www.youtube.com/playlist?list=PLWF9TXck7O_zsqnufs62t26_LJnLo4VRA

Code Execution Instructions

i. Before running this code, please execute command **pip install pyamaze,** if this is not executed before
ii. Unzip file code.zip
iii. Please ensure following csv files are present in same directory as python notebooks
   - Maze_20X20.csv
   - Maze_30X30.csv
   - Maze_40X40.csv
iv. Execute each of python notebooks provided in code folder implementing different maze search algorithms

Name: Ketan Patil
Student Id: 22303876

Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE

Appendix: Code for DFS

```
from pyamaze import maze, agent, COLOR, textLabel
import tracemalloc as memory_trace
import time
from IPython.display import display
import pandas as pd

class Depth_First_Search :

    def __init__(self, maze_size) :
        self.maze_size = maze_size

    def load_maze(self) :
        m = maze()
        maze_name = 'Maze_' + str(self.maze_size) + 'X' + str(self.maze_size)
        m.CreateMaze(loadMaze = maze_name + '.csv')
        return m

    def start_memory_tracing(self) :
        memory_trace.stop()
        memory_trace.start()

    def stop_memory_tracing(self) :
        memory_size, memory_peak = memory_trace.get_traced_memory()
        return memory_size, memory_peak

    def initialise_maze(self) :
        self.maze = self.load_maze()
        self.goal_node = self.maze._goal
        self.start_node = (self.maze_size, self.maze_size)

    def execute_depth_first_search(self):
        self.initialise_maze()
        visited_nodes = [self.start_node]
        stack_next_available_node = [self.start_node]

        explored_nodes = []
        path_traversed = {}

        start_time = time.time() * 1000
        self.start_memory_tracing()
        while len(stack_next_available_node) > 0 :

            current_node = stack_next_available_node.pop()
            explored_nodes.append(current_node)

            if current_node == self.goal_node:
                break

            for __direction__ in ['N', 'S', 'E', 'W']:

                if self.maze.maze_map[current_node][__direction__] == 1 :

                    if __direction__ == 'N' :
                        next_node = (current_node[0] - 1, current_node[1])

                    elif __direction__ == 'S' :
                        next_node = (current_node[0] + 1, current_node[1])

                    elif __direction__ == 'E' :
                        next_node = (current_node[0], current_node[1] + 1)

                    elif __direction__ == 'W' :
                        next_node = (current_node[0], current_node[1] - 1)
```

```
            if next_node in visited_nodes:
                continue

            else:
                stack_next_available_node.append(next_node)
                visited_nodes.append(next_node)
                path_traversed[next_node] = current_node

        end_time = time.time() * 1000
        time_taken = (end_time - start_time)

        memory_size, memory_peak = self.stop_memory_tracing()
        memory_consumed = round((memory_peak/(1024*1024)), 3)

        goal_nodes = self.find_goal_nodes(path_traversed, self.start_node, self.goal_node)


        statistics_df = pd.DataFrame(columns=['Maze Size', 'Time Taken (in ms)', 'Memory Consumed (in MB)', 'Number of Cells Explored',
'Number of Cell in Shortest Path to Goal'])
        statistics_dict = {}
        statistics_dict['Maze Size'] = str(self.maze_size) + 'X' + str(self.maze_size)
        statistics_dict['Time Taken (in ms)'] = time_taken
        statistics_dict['Memory Consumed (in MB)'] = memory_consumed
        statistics_dict['Number of Cells Explored'] = len(path_traversed) + 1
        statistics_dict['Number of Cell in Shortest Path to Goal'] = len(goal_nodes) + 1

        self.display_dfs_path(explored_nodes, goal_nodes, time_taken, memory_consumed, len(path_traversed) + 1, len(goal_nodes) + 1)

        statistics_df = statistics_df.append(statistics_dict, ignore_index = True)

        return statistics_df


    def find_goal_nodes(self, path_traversed, start_node, goal_node) :
        goal_nodes = {}

        while goal_node != start_node :
            goal_nodes[path_traversed[goal_node]] = goal_node
            goal_node = path_traversed[goal_node]

        return goal_nodes

    def display_dfs_path(self, explored_nodes, goal_nodes, time_taken, memory_consumed, len_path_traversed, len_goal_nodes) :
        explored_path = agent(self.maze, x = self.maze_size, y = self.maze_size, goal = (1, 1), footprints = True, color=COLOR.red, filled = True)
        goal_path = agent(self.maze, x = self.maze_size, y = self.maze_size, footprints = True, color=COLOR.cyan)

        self.maze.tracePath({explored_path : explored_nodes}, delay = 10)
        self.maze.tracePath({goal_path : goal_nodes}, delay = 100)

        textLabel(self.maze, 'Maze Size ', str(self.maze_size) + 'X' + str(self.maze_size))
        textLabel(self.maze, 'Time Taken (in ms) ', time_taken)
        textLabel(self.maze, 'Memory Consumed (in MB) ', memory_consumed)
        textLabel(self.maze, 'Number of Cells Explored ', len_path_traversed)
        textLabel(self.maze, 'Number of Cell in Shortest Path to Goal ', len_goal_nodes)

        self.maze.run()

dfs_20 = Depth_First_Search(20)

statistics = dfs_20.execute_depth_first_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)
```

dfs_30 = Depth_First_Search(30)

statistics = dfs_30.execute_depth_first_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

dfs_40 = Depth_First_Search(40)

statistics = dfs_40.execute_depth_first_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

Name: Ketan Patil
Student Id: 22303876

Course Code: CS7IS2-202223 ARTIFICAL INTELLIGENCE

Appendix: Code for BFS

```
from pyamaze import maze, agent, COLOR, textLabel
import tracemalloc as memory_trace
import time
from IPython.display import display
import pandas as pd

class Breadth_First_Search :

  def __init__(self, maze_size) :
    self.maze_size = maze_size

  def load_maze(self) :
    m = maze()
    maze_name = 'Maze_' + str(self.maze_size) + 'X' + str(self.maze_size)
    m.CreateMaze(loadMaze = maze_name + '.csv')
    return m

  def start_memory_tracing(self) :
    memory_trace.stop()
    memory_trace.start()

  def stop_memory_tracing(self) :
    memory_size, memory_peak = memory_trace.get_traced_memory()
    return memory_size, memory_peak

  def initialise_maze(self) :
    self.maze = self.load_maze()
    self.goal_node = self.maze._goal
    self.start_node = (self.maze_size, self.maze_size)

  def execute_breadth_first_search(self):
    self.initialise_maze()
    visited_nodes = [self.start_node]
    stack_next_available_node = [self.start_node]

    explored_nodes = []
    path_traversed = {}

    start_time = time.time() * 1000
    self.start_memory_tracing()
    while len(stack_next_available_node) > 0 :

      current_node = stack_next_available_node.pop(0)
      explored_nodes.append(current_node)

      if current_node == self.goal_node:
        break

      for __direction__ in ['N', 'S', 'E', 'W']:

        if self.maze.maze_map[current_node][__direction__] == 1 :

          if __direction__ == 'N' :
            next_node = (current_node[0] - 1, current_node[1])

          elif __direction__ == 'S' :
            next_node = (current_node[0] + 1, current_node[1])

          elif __direction__ == 'E' :
            next_node = (current_node[0], current_node[1] + 1)

          elif __direction__ == 'W' :
            next_node = (current_node[0], current_node[1] - 1)
```

```
                if next_node in visited_nodes:
                    continue

                else:
                    stack_next_available_node.append(next_node)
                    visited_nodes.append(next_node)
                    path_traversed[next_node] = current_node

        end_time = time.time() * 1000
        time_taken = (end_time - start_time)

        memory_size, memory_peak = self.stop_memory_tracing()
        memory_consumed = round((memory_peak/(1024*1024)), 3)

        goal_nodes = self.find_goal_nodes(path_traversed, self.start_node, self.goal_node)

        statistics_df = pd.DataFrame(columns=['Maze Size', 'Time Taken (in ms)', 'Memory Consumed (in MB)', 'Number of Cells Explored',
'Number of Cell in Shortest Path to Goal'])
        statistics_dict = {}
        statistics_dict['Maze Size'] = str(self.maze_size) + 'X' + str(self.maze_size)
        statistics_dict['Time Taken (in ms)'] = time_taken
        statistics_dict['Memory Consumed (in MB)'] = memory_consumed
        statistics_dict['Number of Cells Explored'] = len(path_traversed) + 1
        statistics_dict['Number of Cell in Shortest Path to Goal'] = len(goal_nodes) + 1

        self.display_bfs_path(explored_nodes, goal_nodes, time_taken, memory_consumed, len(path_traversed) + 1, len(goal_nodes) + 1)

        statistics_df = statistics_df.append(statistics_dict, ignore_index = True)

        return statistics_df


    def find_goal_nodes(self, path_traversed, start_node, goal_node) :
        goal_nodes = {}

        while goal_node != start_node :
            goal_nodes[path_traversed[goal_node]] = goal_node
            goal_node = path_traversed[goal_node]

        return goal_nodes

    def display_bfs_path(self, explored_nodes, goal_nodes, time_taken, memory_consumed, len_path_traversed, len_goal_nodes) :
        explored_path = agent(self.maze, x = self.maze_size, y = self.maze_size, goal = (1, 1), footprints = True, color=COLOR.red, filled = True)
        goal_path = agent(self.maze, x = self.maze_size, y = self.maze_size, footprints = True, color=COLOR.cyan)

        self.maze.tracePath({explored_path : explored_nodes}, delay = 10)
        self.maze.tracePath({goal_path : goal_nodes}, delay = 100)

        textLabel(self.maze, 'Maze Size ', str(self.maze_size) + 'X' + str(self.maze_size))
        textLabel(self.maze, 'Time Taken (in ms) ', time_taken)
        textLabel(self.maze, 'Memory Consumed (in MB) ', memory_consumed)
        textLabel(self.maze, 'Number of Cells Explored ', len_path_traversed)
        textLabel(self.maze, 'Number of Cell in Shortest Path to Goal ', len_goal_nodes)

        self.maze.run()

bfs_20 = Breadth_First_Search(20)

statistics = bfs_20.execute_breadth_first_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)
```

bfs_30 = Breadth_First_Search(30)

statistics = bfs_30.execute_breadth_first_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

bfs_40 = Breadth_First_Search(40)

statistics = bfs_40.execute_breadth_first_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

Name: Ketan Patil
Student Id: 22303876

Appendix: Code for A* using Euclidean Distance

```
from pyamaze import maze, agent, COLOR, textLabel
import tracemalloc as memory_trace
import time
from IPython.display import display
import pandas as pd
from queue import PriorityQueue
import numpy as np

class A_Star_Search :

  def __init__(self, maze_size) :
    self.maze_size = maze_size

  def load_maze(self) :
    m = maze()
    maze_name = 'Maze_' + str(self.maze_size) + 'X' + str(self.maze_size)
    m.CreateMaze(loadMaze = maze_name + '.csv')
    return m

  def start_memory_tracing(self) :
    memory_trace.stop()
    memory_trace.start()

  def stop_memory_tracing(self) :
    memory_size, memory_peak = memory_trace.get_traced_memory()
    return memory_size, memory_peak

  def initialize_maze(self) :
    self.maze = self.load_maze()
    self.goal_node = self.maze._goal
    self.start_node = (self.maze_size, self.maze_size)

  def initialise_cost(self) :
    self.next_node_cost = {node : 0 if node == self.start_node else float('inf') for node in self.maze.grid}
    self.total_cost = {node :  0 + self.get_euclidian_distance_heuristic_cost(self.start_node) if node == self.start_node else float('inf') for
node in self.maze.grid}


  def execute_a_star_search(self):
    self.initialize_maze()
    self.initialise_cost()
    priority_queue = PriorityQueue()
    priority_queue.put((0 + self.get_euclidian_distance_heuristic_cost(self.start_node),
self.get_euclidian_distance_heuristic_cost(self.start_node), self.start_node))

    explored_nodes = []
    path_traversed = {}

    start_time = time.time() * 1000
    self.start_memory_tracing()
    while not priority_queue.empty() :

      current_node = priority_queue.get()[2]
      explored_nodes.append(current_node)

      if current_node == self.goal_node:
        break

      for __direction__ in ['N', 'S', 'E', 'W']:

        if self.maze.maze_map[current_node][__direction__] == 1 :

          if __direction__ == 'N' :
            next_node = (current_node[0] - 1, current_node[1])
```

```
            elif __direction__ == 'S' :
                next_node = (current_node[0] + 1, current_node[1])

            elif __direction__ == 'E' :
                next_node = (current_node[0], current_node[1] + 1)

            elif __direction__ == 'W' :
                next_node = (current_node[0], current_node[1] - 1)

            var_next_node_cost = self.next_node_cost[current_node] + 1
            var_total_cost = var_next_node_cost + self.get_euclidian_distance_heuristic_cost(next_node)

            if var_total_cost < self.total_cost[next_node] :
                self.total_cost[next_node] = var_total_cost
                self.next_node_cost[next_node] = var_next_node_cost
                priority_queue.put((var_total_cost, self.get_euclidian_distance_heuristic_cost(next_node), next_node))
                path_traversed[next_node] = current_node

        end_time = time.time() * 1000
        time_taken = (end_time - start_time)

        memory_size, memory_peak = self.stop_memory_tracing()
        memory_consumed = round((memory_peak/(1024*1024)), 3)

        goal_nodes = self.find_goal_nodes(path_traversed, self.start_node, self.goal_node)

        statistics_df = pd.DataFrame(columns=['Maze Size', 'Time Taken (in ms)', 'Memory Consumed (in MB)', 'Number of Cells Explored',
'Number of Cell in Shortest Path to Goal'])
        statistics_dict = {}
        statistics_dict['Maze Size'] = str(self.maze_size) + 'X' + str(self.maze_size)
        statistics_dict['Time Taken (in ms)'] = time_taken
        statistics_dict['Memory Consumed (in MB)'] = memory_consumed
        statistics_dict['Number of Cells Explored'] = len(path_traversed) + 1
        statistics_dict['Number of Cell in Shortest Path to Goal'] = len(goal_nodes) + 1

        self.display_astar_path(explored_nodes, goal_nodes, time_taken, memory_consumed, len(path_traversed) + 1, len(goal_nodes) + 1)

        statistics_df = statistics_df.append(statistics_dict, ignore_index = True)

        return statistics_df

    def get_euclidian_distance_heuristic_cost(self, node):
        x, y = node
        goal_x, goal_y = self.maze._goal
        return np.sqrt(pow((goal_x - x), 2) + pow((goal_y - y), 2))

    def find_goal_nodes(self, path_traversed, start_node, goal_node) :
        goal_nodes = {}

        while goal_node != start_node :
            goal_nodes[path_traversed[goal_node]] = goal_node
            goal_node = path_traversed[goal_node]

        return goal_nodes

    def display_astar_path(self, explored_nodes, goal_nodes, time_taken, memory_consumed, len_path_traversed, len_goal_nodes) :
        explored_path = agent(self.maze, x = self.maze_size, y = self.maze_size, goal = (1, 1), footprints = True, color=COLOR.red, filled = True)
        goal_path = agent(self.maze, x = self.maze_size, y = self.maze_size, footprints = True, color=COLOR.cyan)

        self.maze.tracePath({explored_path : explored_nodes}, delay = 10)
        self.maze.tracePath({goal_path : goal_nodes}, delay = 100)

        textLabel(self.maze, 'Maze Size ', str(self.maze_size) + 'X' + str(self.maze_size))
        textLabel(self.maze, 'Time Taken (in ms) ', time_taken)
```

```
    textLabel(self.maze, 'Memory Consumed (in MB) ', memory_consumed)
    textLabel(self.maze, 'Number of Cells Explored ', len_path_traversed)
    textLabel(self.maze, 'Number of Cell in Shortest Path to Goal ', len_goal_nodes)

    self.maze.run()

astar_20 = A_Star_Search(20)

statistics = astar_20.execute_a_star_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

astar_30 = A_Star_Search(30)

statistics = astar_30.execute_a_star_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

astar_40 = A_Star_Search(40)

statistics = astar_40.execute_a_star_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)
```

Name: Ketan Patil
Student Id: 22303876

Appendix: Code for A* using Manhattan Distance

```
from pyamaze import maze, agent, COLOR, textLabel
import tracemalloc as memory_trace
import time
from IPython.display import display
import pandas as pd
from queue import PriorityQueue

class A_Star_Search :

    def __init__(self, maze_size) :
        self.maze_size = maze_size

    def load_maze(self) :
        m = maze()
        maze_name = 'Maze_' + str(self.maze_size) + 'X' + str(self.maze_size)
        m.CreateMaze(loadMaze = maze_name + '.csv')
        return m

    def start_memory_tracing(self) :
        memory_trace.stop()
        memory_trace.start()

    def stop_memory_tracing(self) :
        memory_size, memory_peak = memory_trace.get_traced_memory()
        return memory_size, memory_peak

    def initialize_maze(self) :
        self.maze = self.load_maze()
        self.goal_node = self.maze._goal
        self.start_node = (self.maze_size, self.maze_size)

    def initialise_cost(self) :
        self.next_node_cost = {node : 0 if node == self.start_node else float('inf') for node in self.maze.grid}
        self.total_cost = {node :  0 + self.get_manhattan_distance_heuristic_cost(self.start_node) if node == self.start_node else float('inf') for
node in self.maze.grid}


    def execute_a_star_search(self):
        self.initialize_maze()
        self.initialise_cost()
        priority_queue = PriorityQueue()
        priority_queue.put((0 + self.get_manhattan_distance_heuristic_cost(self.start_node),
self.get_manhattan_distance_heuristic_cost(self.start_node), self.start_node))

        explored_nodes = []
        path_traversed = {}

        start_time = time.time() * 1000
        self.start_memory_tracing()
        while not priority_queue.empty() :

            current_node = priority_queue.get()[2]
            explored_nodes.append(current_node)

            if current_node == self.goal_node:
                break

            for __direction__ in ['N', 'S', 'E', 'W']:

                if self.maze.maze_map[current_node][__direction__] == 1 :

                    if __direction__ == 'N' :
                        next_node = (current_node[0] - 1, current_node[1])
```

```
            elif __direction__ == 'S' :
                next_node = (current_node[0] + 1, current_node[1])

            elif __direction__ == 'E' :
                next_node = (current_node[0], current_node[1] + 1)

            elif __direction__ == 'W' :
                next_node = (current_node[0], current_node[1] - 1)

            var_next_node_cost = self.next_node_cost[current_node] + 1
            var_total_cost = var_next_node_cost + self.get_manhattan_distance_heuristic_cost(next_node)

            if var_total_cost < self.total_cost[next_node] :
                self.total_cost[next_node] = var_total_cost
                self.next_node_cost[next_node] = var_next_node_cost
                priority_queue.put((var_total_cost, self.get_manhattan_distance_heuristic_cost(next_node), next_node))
                path_traversed[next_node] = current_node

        end_time = time.time() * 1000
        time_taken = (end_time - start_time)

        memory_size, memory_peak = self.stop_memory_tracing()
        memory_consumed = round((memory_peak/(1024*1024)), 3)

        goal_nodes = self.find_goal_nodes(path_traversed, self.start_node, self.goal_node)

        statistics_df = pd.DataFrame(columns=['Maze Size', 'Time Taken (in ms)', 'Memory Consumed (in MB)', 'Number of Cells Explored',
'Number of Cell in Shortest Path to Goal'])
        statistics_dict = {}
        statistics_dict['Maze Size'] = str(self.maze_size) + 'X' + str(self.maze_size)
        statistics_dict['Time Taken (in ms)'] = time_taken
        statistics_dict['Memory Consumed (in MB)'] = memory_consumed
        statistics_dict['Number of Cells Explored'] = len(path_traversed) + 1
        statistics_dict['Number of Cell in Shortest Path to Goal'] = len(goal_nodes) + 1

        self.display_astar_path(explored_nodes, goal_nodes, time_taken, memory_consumed, len(path_traversed) + 1, len(goal_nodes) + 1)

        statistics_df = statistics_df.append(statistics_dict, ignore_index = True)

        return statistics_df

    def get_manhattan_distance_heuristic_cost(self, node):
        x, y = node
        goal_x, goal_y = self.maze._goal
        return abs(goal_x - x) + abs(goal_y - y)

    def find_goal_nodes(self, path_traversed, start_node, goal_node) :
        goal_nodes = {}

        while goal_node != start_node :
            goal_nodes[path_traversed[goal_node]] = goal_node
            goal_node = path_traversed[goal_node]

        return goal_nodes

    def display_astar_path(self, explored_nodes, goal_nodes, time_taken, memory_consumed, len_path_traversed, len_goal_nodes) :
        explored_path = agent(self.maze, x = self.maze_size, y = self.maze_size, goal = (1, 1), footprints = True, color=COLOR.red, filled = True)
        goal_path = agent(self.maze, x = self.maze_size, y = self.maze_size, footprints = True, color=COLOR.cyan)

        self.maze.tracePath({explored_path : explored_nodes}, delay = 10)
        self.maze.tracePath({goal_path : goal_nodes}, delay = 100)

        textLabel(self.maze, 'Maze Size ', str(self.maze_size) + 'X' + str(self.maze_size))
        textLabel(self.maze, 'Time Taken (in ms) ', time_taken)
        textLabel(self.maze, 'Memory Consumed (in MB) ', memory_consumed)
```

```
    textLabel(self.maze, 'Number of Cells Explored ', len_path_traversed)
    textLabel(self.maze, 'Number of Cell in Shortest Path to Goal ', len_goal_nodes)

    self.maze.run()

astar_20 = A_Star_Search(20)

statistics = astar_20.execute_a_star_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

astar_30 = A_Star_Search(30)

statistics = astar_30.execute_a_star_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

astar_40 = A_Star_Search(40)

statistics = astar_40.execute_a_star_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)
```

Name: Ketan Patil
Student Id: 22303876

Appendix: Code for MDP Value Iteration

```
from pyamaze import maze, agent, COLOR, textLabel
import tracemalloc as memory_trace
import time
from IPython.display import display
import pandas as pd
import copy

class MDP_Value_Iteration_Search :

   def __init__(self, maze_size) :
      self.maze_size = maze_size

   def load_maze(self) :
      m = maze()
      maze_name = 'Maze_' + str(self.maze_size) + 'X' + str(self.maze_size)
      m.CreateMaze(loadMaze = maze_name + '.csv')
      return m

   def start_memory_tracing(self) :
      memory_trace.stop()
      memory_trace.start()

   def stop_memory_tracing(self) :
      memory_size, memory_peak = memory_trace.get_traced_memory()
      return memory_size, memory_peak

   def initialize_maze(self) :
      self.maze = self.load_maze()
      self.goal_node = self.maze._goal
      self.start_node = (self.maze_size, self.maze_size)

   def initialise_cost(self) :
      self.transition_value = {node: 10 if node == self.maze._goal else 0 for node in self.maze.grid}
      self.transition_reward = {node: 100 if node == self.maze._goal else 0 for node in self.maze.grid}

      self.transition_dictionary = copy.deepcopy(self.maze.maze_map)
      for key in self.transition_dictionary :
         for subkey in self.transition_dictionary[key] :
            self.transition_dictionary[key][subkey] = 0

      self.initial_transition_value = {}
      self.initial_transition_value['N'] = 1
      self.initial_transition_value['S'] = 1
      self.initial_transition_value['E'] = 1
      self.initial_transition_value['W'] = 1

      self.gamma = 0.9
      self.threshold = 0.000001

   def execute_mdp_value_iteration_search(self):

      self.initialize_maze()
      self.initialise_cost()

      start_time = time.time() * 1000
      self.start_memory_tracing()
      has_value_converged = False

      while not has_value_converged :
         has_value_converged = True

         for current_node in self.maze.grid :
```

```
            temp_transition_value = []

            for __direction__ in ['N', 'S', 'E', 'W']:

                if self.maze.maze_map[current_node][__direction__] == 1 :

                    try:
                        if __direction__ == 'N' :
                            next_node = (current_node[0] - 1, current_node[1])

                        elif __direction__ == 'S' :
                            next_node = (current_node[0] + 1, current_node[1])

                        elif __direction__ == 'E' :
                            next_node = (current_node[0], current_node[1] + 1)

                        elif __direction__ == 'W' :
                            next_node = (current_node[0], current_node[1] - 1)
                    except:
                        next_node = None

                    if next_node is not None:
                        next_transtion_value = self.initial_transition_value[__direction__] * (self.transition_reward[current_node] +
self.transition_value[next_node] * self.gamma)
                        temp_transition_value.append(next_transtion_value)
                        self.transition_dictionary[current_node][__direction__] = next_transtion_value

            best_transtion_value = (max(temp_transition_value))

            if abs(best_transtion_value - self.transition_value[current_node]) > self.threshold :
                has_value_converged = False
                self.transition_value[current_node] = best_transtion_value

        end_time = time.time() * 1000
        time_taken = (end_time - start_time)

        memory_size, memory_peak = self.stop_memory_tracing()
        memory_consumed = round((memory_peak/(1024*1024)), 3)
        goal_nodes = self.find_goal_nodes(self.transition_dictionary, self.start_node, self.goal_node)

        statistics_df = pd.DataFrame(columns=['Maze Size', 'Time Taken (in ms)', 'Memory Consumed (in MB)', 'Number of Cell in Shortest Path
to Goal'])
        statistics_dict = {}
        statistics_dict['Maze Size'] = str(self.maze_size) + 'X' + str(self.maze_size)
        statistics_dict['Time Taken (in ms)'] = time_taken
        statistics_dict['Memory Consumed (in MB)'] = memory_consumed
        statistics_dict['Number of Cell in Shortest Path to Goal'] = len(goal_nodes) + 1

        self.display_mdp_value_iteration_path(goal_nodes, time_taken, memory_consumed, len(goal_nodes) + 1)

        statistics_df = statistics_df.append(statistics_dict, ignore_index = True)

        return statistics_df


    def find_goal_nodes(self, transition_dictionary, start_node, goal_node) :
        goal_nodes = {}
        next_node_to_goal = [start_node]

        while len(next_node_to_goal) > 0 :
            current_node = next_node_to_goal.pop()

            if current_node == goal_node :
                break
```

```
        best_transition_policy = self.find_best_transition_direction(self.transition_dictionary[current_node])
        print(f'\nCurrent Cell: {current_node}, Best Transition State for this cell: {transition_dictionary[current_node]}, Best Transition
Direction: {best_transition_policy}')
        if best_transition_policy == 'N' :
          next_node = (current_node[0] - 1, current_node[1])

        elif best_transition_policy == 'S' :
          next_node = (current_node[0] + 1, current_node[1])

        elif best_transition_policy == 'E' :
          next_node = (current_node[0], current_node[1] + 1)

        elif best_transition_policy == 'W' :
          next_node = (current_node[0], current_node[1] - 1)

        goal_nodes[current_node] = next_node
        next_node_to_goal.append(next_node)

      return goal_nodes


  def find_best_transition_direction(self, current_node) :
    transition_values = list(current_node.values())
    directions = list(current_node.keys())
    return directions[transition_values.index(max(transition_values))]

  def display_mdp_value_iteration_path(self, goal_nodes, time_taken, memory_consumed, len_goal_nodes) :

    goal_path = agent(self.maze, x = self.maze_size, y = self.maze_size, footprints = True, color=COLOR.cyan)
    self.maze.tracePath({goal_path : goal_nodes}, delay = 100)

    textLabel(self.maze, 'Maze Size ', str(self.maze_size) + 'X' + str(self.maze_size))
    textLabel(self.maze, 'Time Taken (in ms) ', time_taken)
    textLabel(self.maze, 'Memory Consumed (in MB) ', memory_consumed)
    textLabel(self.maze, 'Number of Cell in Shortest Path to Goal ', len_goal_nodes)

    self.maze.run()

mdp_value_iteration_20 = MDP_Value_Iteration_Search(20)

statistics = mdp_value_iteration_20.execute_mdp_value_iteration_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

mdp_value_iteration_30 = MDP_Value_Iteration_Search(30)

statistics = mdp_value_iteration_30.execute_mdp_value_iteration_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

mdp_value_iteration_40 = MDP_Value_Iteration_Search(40)

statistics = mdp_value_iteration_40.execute_mdp_value_iteration_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)
```

Name: Ketan Patil
Student Id: 22303876

Appendix: Code for MDP Policy Iteration

```
from pyamaze import maze, agent, COLOR, textLabel
import tracemalloc as memory_trace
import time
from IPython.display import display
import pandas as pd
import copy

class MDP_Policy_Iteration_Search :

  def __init__(self, maze_size) :
    self.maze_size = maze_size

  def load_maze(self) :
    m = maze()
    maze_name = 'Maze_' + str(self.maze_size) + 'X' + str(self.maze_size)
    m.CreateMaze(loadMaze = maze_name + '.csv')
    return m

  def start_memory_tracing(self) :
    memory_trace.stop()
    memory_trace.start()

  def stop_memory_tracing(self) :
    memory_size, memory_peak = memory_trace.get_traced_memory()
    return memory_size, memory_peak

  def initialize_maze(self) :
    self.maze = self.load_maze()
    self.goal_node = self.maze._goal
    self.start_node = (self.maze_size, self.maze_size)

  def initialise_cost(self) :
    self.transition_value = {node: 1 if node == self.maze._goal else 0 for node in self.maze.grid}
    self.transition_reward = {node: 1 if node == self.maze._goal else 0 for node in self.maze.grid}

    self.transition_dictionary = copy.deepcopy(self.maze.maze_map)
    for key in self.transition_dictionary :
      for subkey in self.transition_dictionary[key] :
        self.transition_dictionary[key][subkey] = 0

    self.transition_policy = {node: None if node == self.maze._goal else 'N' for node in self.maze.grid}
    self.initial_transition_value = {}
    self.initial_transition_value['N'] = 1
    self.initial_transition_value['S'] = 1
    self.initial_transition_value['E'] = 1
    self.initial_transition_value['W'] = 1

    self.gamma = 0.9
    self.threshold = 0.000001

  def execute_value_iteration(self, current_node) :
    temporary_transition_value = {}
    temporary_transition_value['N'] = 0
    temporary_transition_value['S'] = 0
    temporary_transition_value['E'] = 0
    temporary_transition_value['W'] = 0
    temporary_node_transtion_value = {current_node : temporary_transition_value}

    for __direction__ in ['N', 'S', 'E', 'W']:

      if self.maze.maze_map[current_node][__direction__] == 1 :

        try:
          if __direction__ == 'N' :
```

```
                next_node = (current_node[0] - 1, current_node[1])

              elif __direction__ == 'S' :
                next_node = (current_node[0] + 1, current_node[1])

              elif __direction__ == 'E' :
                next_node = (current_node[0], current_node[1] + 1)

              elif __direction__ == 'W' :
                next_node = (current_node[0], current_node[1] - 1)
            except:
              next_node = None

            if next_node is not None :
              next_node_value = self.transition_value[next_node]
            else :
              next_node_value = 0

            temporary_node_transtion_value[current_node][__direction__] =  self.initial_transition_value[__direction__] *
(self.transition_reward[current_node] + next_node_value * self.gamma)

        return temporary_node_transtion_value


    def execute_mdp_policy_iteration_search(self):

        self.initialize_maze()
        self.initialise_cost()

        start_time = time.time() * 1000
        self.start_memory_tracing()
        has_value_converged = False
        has_policy_converged = False

        while not has_policy_converged :
          has_policy_converged = True

          has_value_converged = False
          while not has_value_converged :
            has_value_converged = True

            for current_node in self.maze.grid :

              if current_node == self.goal_node :
                continue

              current_policy = self.transition_policy[current_node]
              if self.maze.maze_map[current_node][current_policy] == 1 :
                try:
                  if current_policy == 'N' :
                    next_node = (current_node[0] - 1, current_node[1])

                  elif current_policy == 'S' :
                    next_node = (current_node[0] + 1, current_node[1])

                  elif current_policy == 'E' :
                    next_node = (current_node[0], current_node[1] + 1)

                  elif current_policy == 'W' :
                    next_node = (current_node[0], current_node[1] - 1)
                except:
                  next_node = None

                if next_node is not None :
                  next_node_value = self.transition_value[next_node]
```

```
                else :
                    next_node_value = 0

                self.transition_dictionary[current_node][current_policy] =  self.initial_transition_value[current_policy] *
(self.transition_reward[current_node] + next_node_value * self.gamma)

                if abs(self.transition_value[current_node] - (self.initial_transition_value[current_policy] *
(self.transition_reward[current_node] + next_node_value * self.gamma))) > self.threshold :
                    self.transition_value[current_node] = self.initial_transition_value[current_policy] * (self.transition_reward[current_node] +
next_node_value * self.gamma)
                    has_value_converged = False

          for current_node in self.maze.grid :

            if current_node == self.goal_node :
              continue

            current_node_transition_value = self.execute_value_iteration(current_node)

            current_node_transition_policy = self.find_best_transition_direction(current_node_transition_value[current_node])

            if self.transition_policy[current_node] != current_node_transition_policy :
              self.transition_policy[current_node] = current_node_transition_policy
              has_policy_converged = False

        end_time = time.time() * 1000
        time_taken = (end_time - start_time)

        memory_size, memory_peak = self.stop_memory_tracing()
        memory_consumed = round((memory_peak/(1024*1024)), 3)
        goal_nodes = self.find_goal_nodes(self.transition_policy, self.start_node, self.goal_node)

        statistics_df = pd.DataFrame(columns=['Maze Size', 'Time Taken (in ms)', 'Memory Consumed (in MB)', 'Number of Cell in Shortest Path
to Goal'])
        statistics_dict = {}
        statistics_dict['Maze Size'] = str(self.maze_size) + 'X' + str(self.maze_size)
        statistics_dict['Time Taken (in ms)'] = time_taken
        statistics_dict['Memory Consumed (in MB)'] = memory_consumed
        statistics_dict['Number of Cell in Shortest Path to Goal'] = len(goal_nodes) + 1

        self.display_mdp_value_iteration_path(goal_nodes, time_taken, memory_consumed, len(goal_nodes) + 1)

        statistics_df = statistics_df.append(statistics_dict, ignore_index = True)

        return statistics_df

    def find_goal_nodes(self, transition_policy, start_node, goal_node) :
        goal_nodes = {}
        next_node_to_goal = [start_node]

        while len(next_node_to_goal) > 0 :
          current_node = next_node_to_goal.pop()

          if current_node == goal_node :
            break

          best_transition_policy = transition_policy[current_node]
          print(f'\nCurrent Cell: {current_node}, Best Transition Direction: {best_transition_policy}')
          if best_transition_policy == 'N' :
            next_node = (current_node[0] - 1, current_node[1])

          elif best_transition_policy == 'S' :
            next_node = (current_node[0] + 1, current_node[1])

          elif best_transition_policy == 'E' :
```

```
            next_node = (current_node[0], current_node[1] + 1)

        elif best_transition_policy == 'W' :
            next_node = (current_node[0], current_node[1] - 1)

        goal_nodes[current_node] = next_node
        next_node_to_goal.append(next_node)

    return goal_nodes

def find_best_transition_direction(self, current_node) :
    transition_values = list(current_node.values())
    directions = list(current_node.keys())
    return directions[transition_values.index(max(transition_values))]

def display_mdp_value_iteration_path(self, goal_nodes, time_taken, memory_consumed, len_goal_nodes) :

    goal_path = agent(self.maze, x = self.maze_size, y = self.maze_size, footprints = True, color=COLOR.cyan)
    self.maze.tracePath({goal_path : goal_nodes}, delay = 100)

    textLabel(self.maze, 'Maze Size ', str(self.maze_size) + 'X' + str(self.maze_size))
    textLabel(self.maze, 'Time Taken (in ms) ', time_taken)
    textLabel(self.maze, 'Memory Consumed (in MB) ', memory_consumed)
    textLabel(self.maze, 'Number of Cell in Shortest Path to Goal ', len_goal_nodes)

    self.maze.run()

mdp_policy_iteration_20 = MDP_Policy_Iteration_Search(20)

statistics = mdp_policy_iteration_20.execute_mdp_policy_iteration_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

mdp_policy_iteration_30 = MDP_Policy_Iteration_Search(30)

statistics = mdp_policy_iteration_30.execute_mdp_policy_iteration_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)

mdp_policy_iteration_40 = MDP_Policy_Iteration_Search(40)

statistics = mdp_policy_iteration_40.execute_mdp_policy_iteration_search()

statistics = statistics.style.applymap(lambda x:'white-space:nowrap')
display(statistics)
```

Appendix: Pyamaze License

1. Module Code

https://github.com/MAN1986/pyamaze/blob/main/pyamaze/pyamaze.py

2. License Statement

```
License
          https://www.youtube.com/c/LearningOrbis
          Copyright (c) 2021 Muhammad Ahsan Naeem
          mahsan.naeem@gmail.com
          Permission is hereby granted, free of charge, to any person obtaining a copy
          of this software and associated documentation files (the "Software"), to deal
          in the Software without restriction, including without limitation the rights
          to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
          copies of the Software, and to permit persons to whom the Software is
          furnished to do so, subject to the following conditions:
          The above copyright notice and this permission notice shall be included in all
          copies or substantial portions of the Software.
          THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
          IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
          FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
          AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
          LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
          OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
          SOFTWARE.
          """
```