**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:

1. 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.
2. 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.

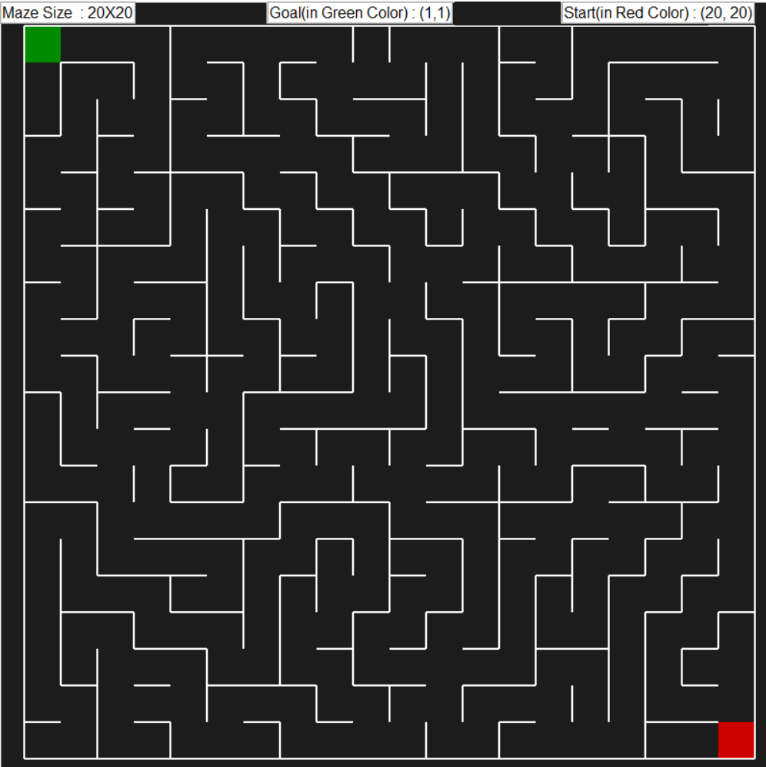
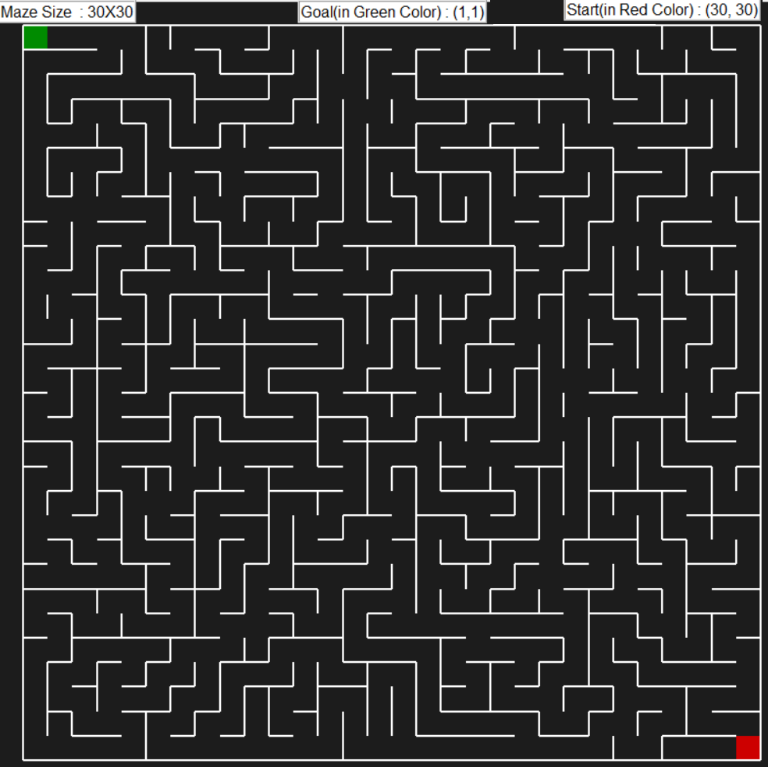
Furthermore, this API also provides simple functions to generate visualisations for mazes and provide features to track the path of algorithms as it traverses the maze.

1. **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:

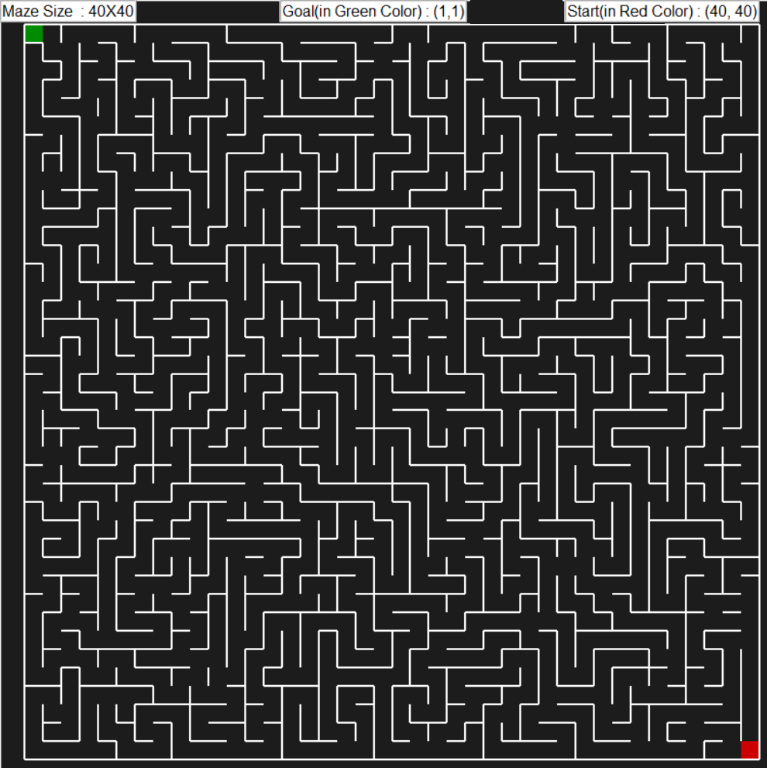


Figure 1: Maze of size 20X20, 30X30, and 40X40

1. **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:

1. Manhattan Distance
2. Euclidean Distance
3. **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.

**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.

1. **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.

1. **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.

1. **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.

1. **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

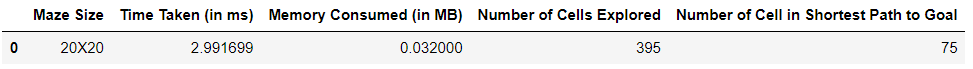
Furthermore, A\* algorithm with Manhattan distance as heuristic outperforms Euclidean distance for all maze sizes.

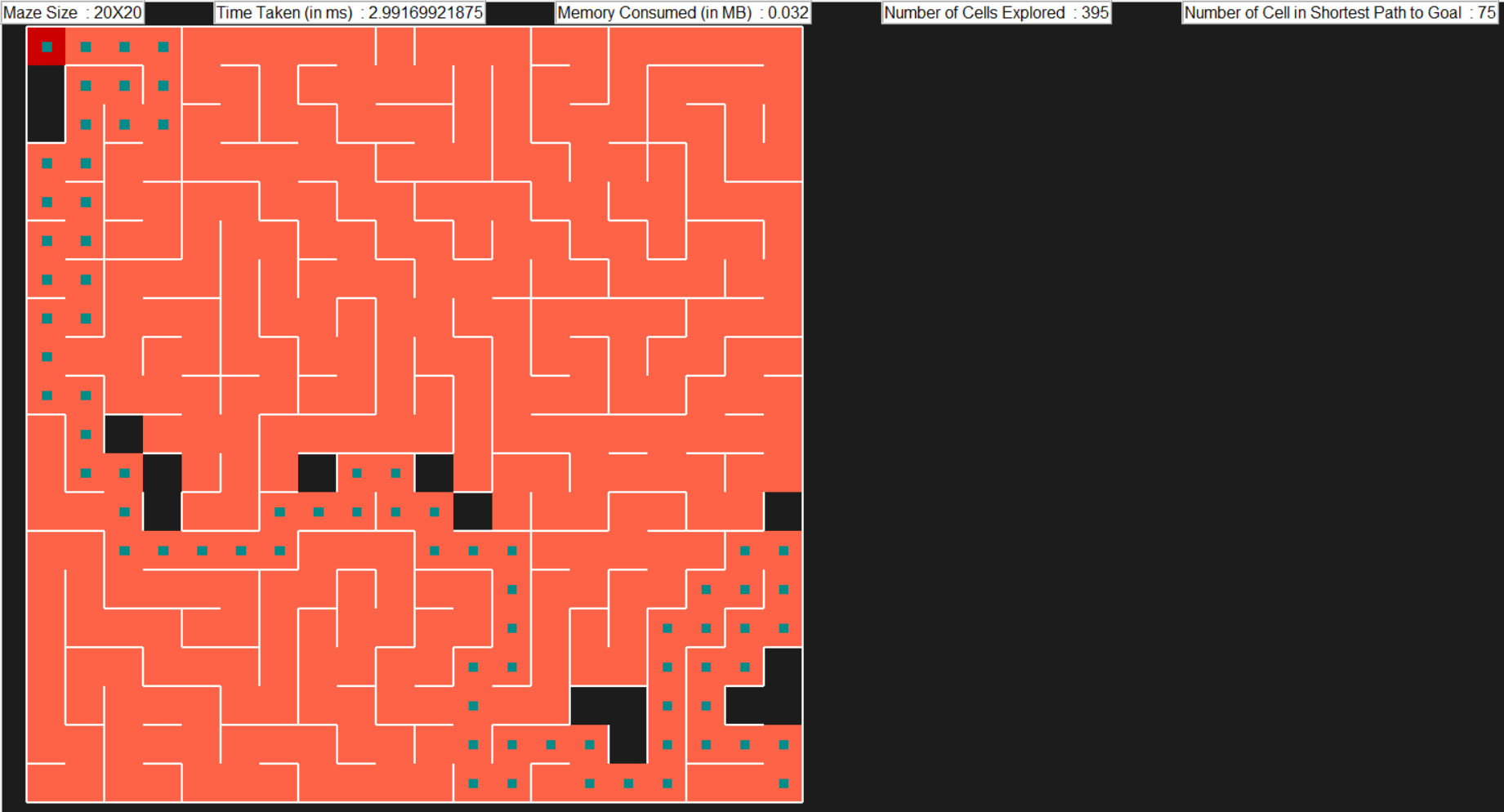
**Conclusion**

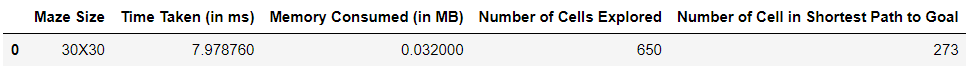
I can conclude that behaviour of these algorithms varies with maze size therefore, we should analyse the complexity of the problem before applying these algorithms. Performance of Search and MDP based algorithms are very different due to the number of evaluations involved in each step. MDP algorithms 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**

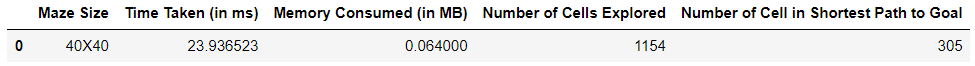
1. **Depth First Search**

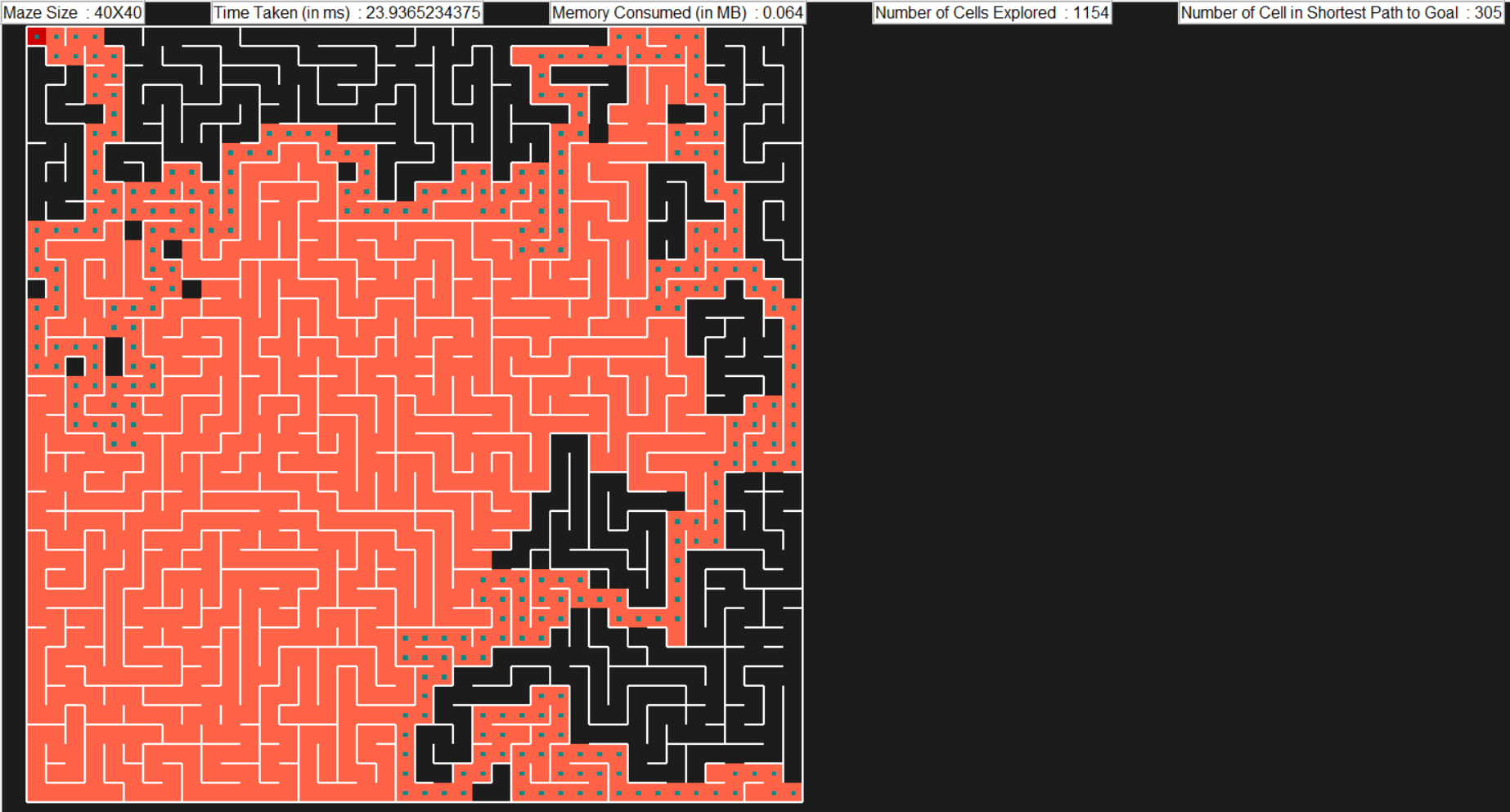
****

****

****

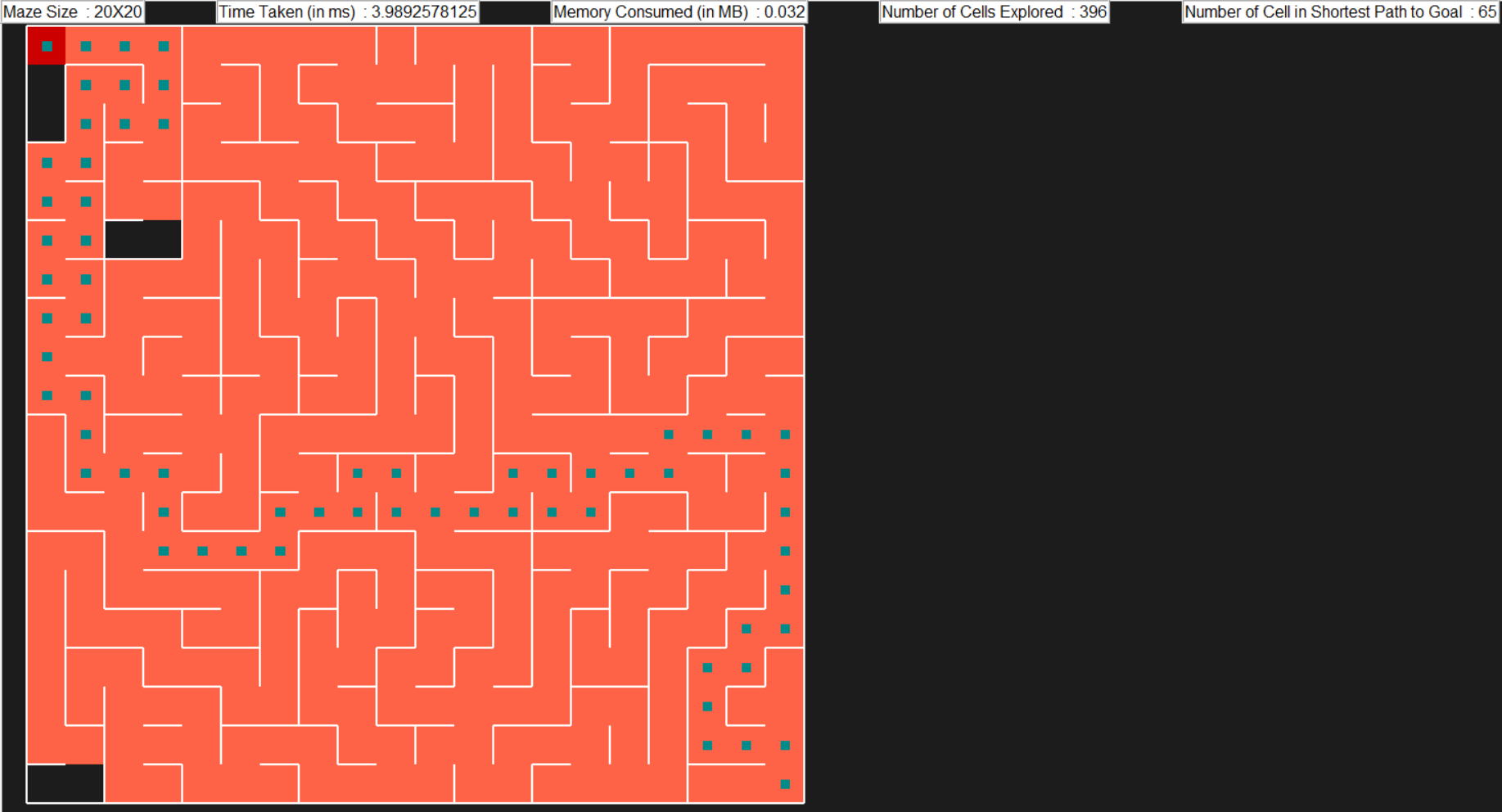
****

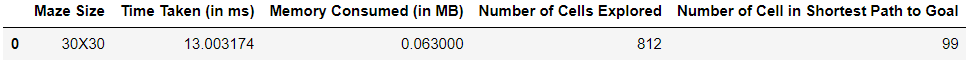
****

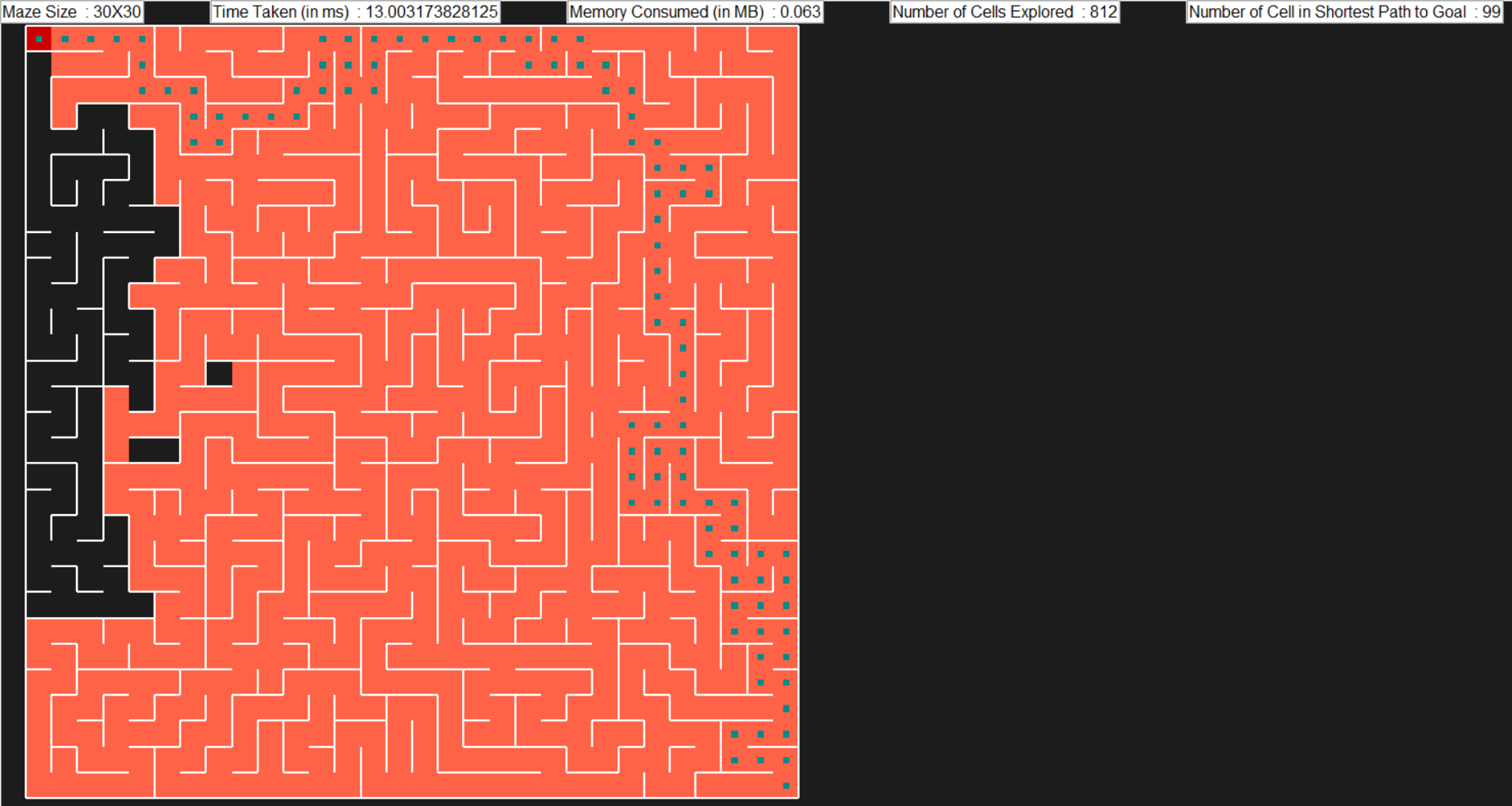
****

1. **Breadth First Search**

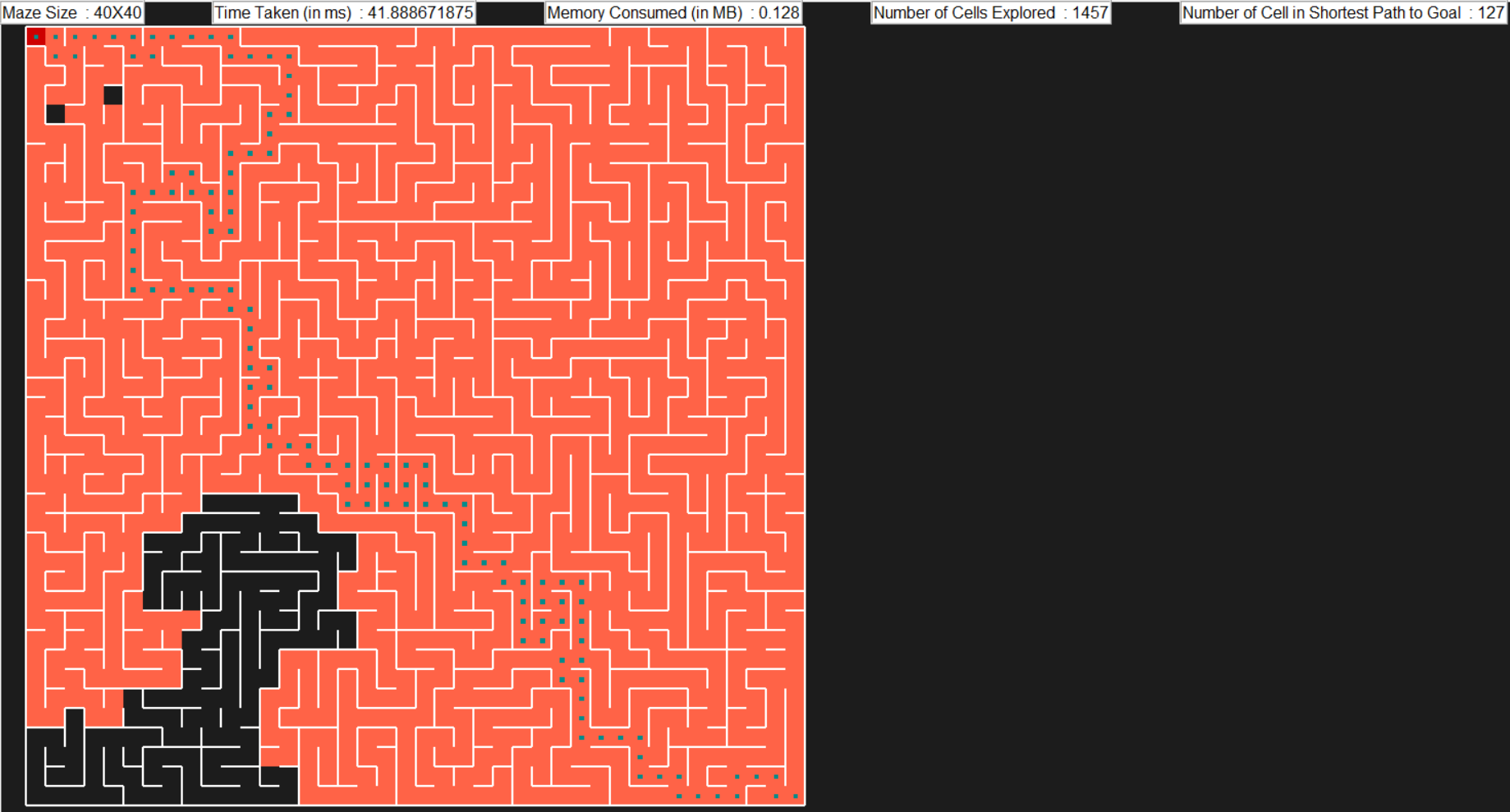
****

****

****

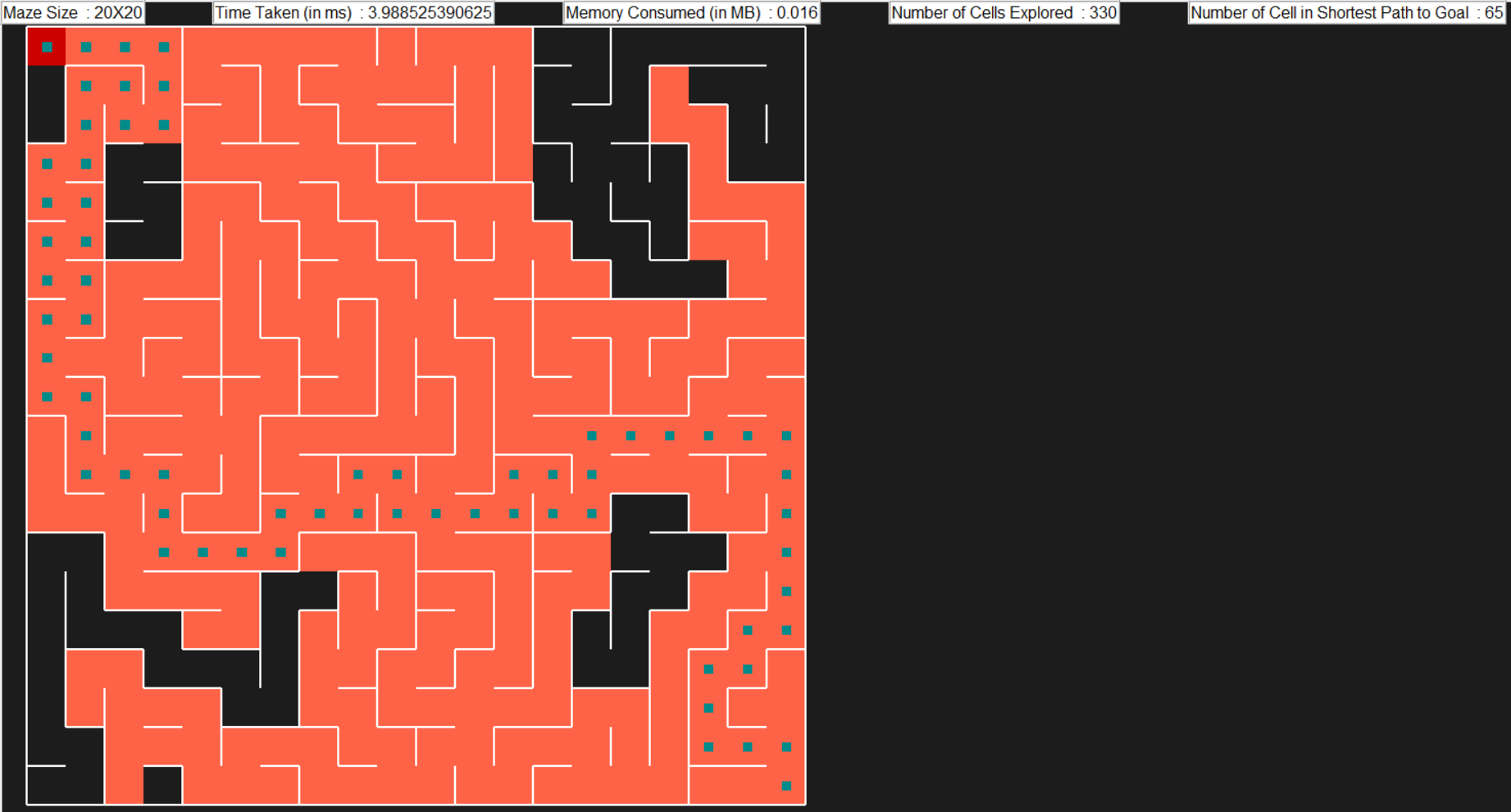
****

****

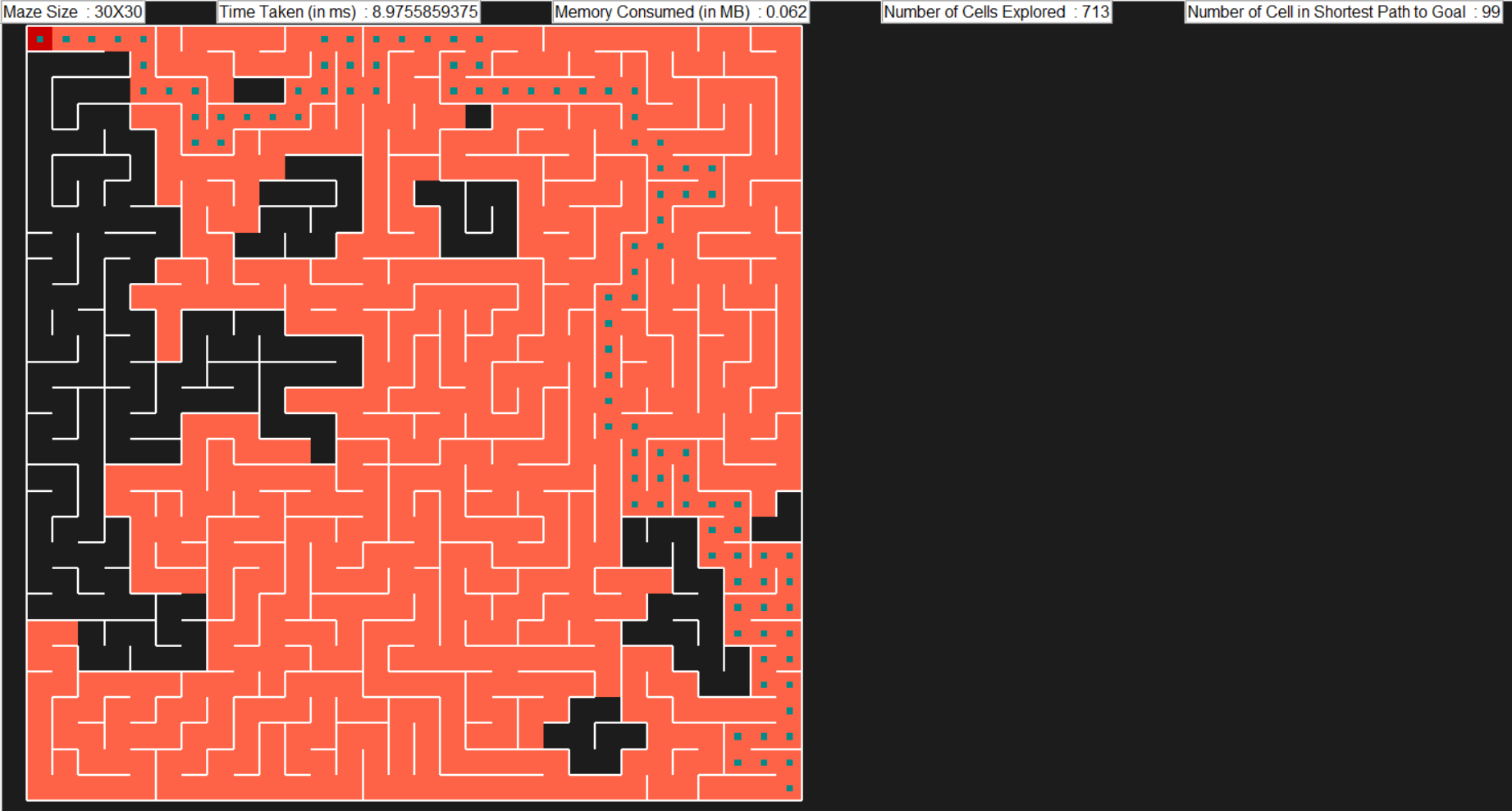
****

1. **A Star Search with Manhattan Distance as Heuristics**

****

****

****

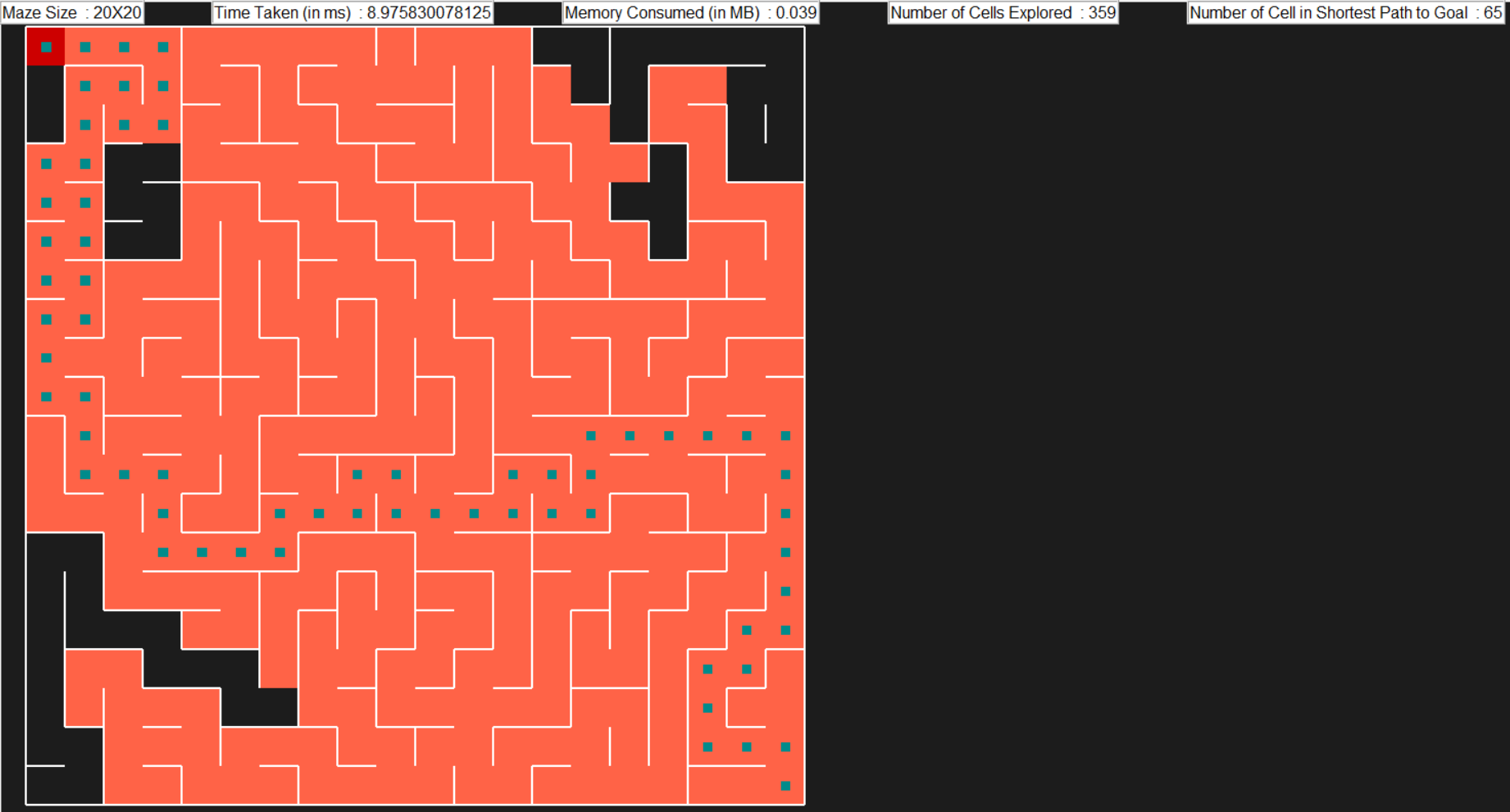
****

****

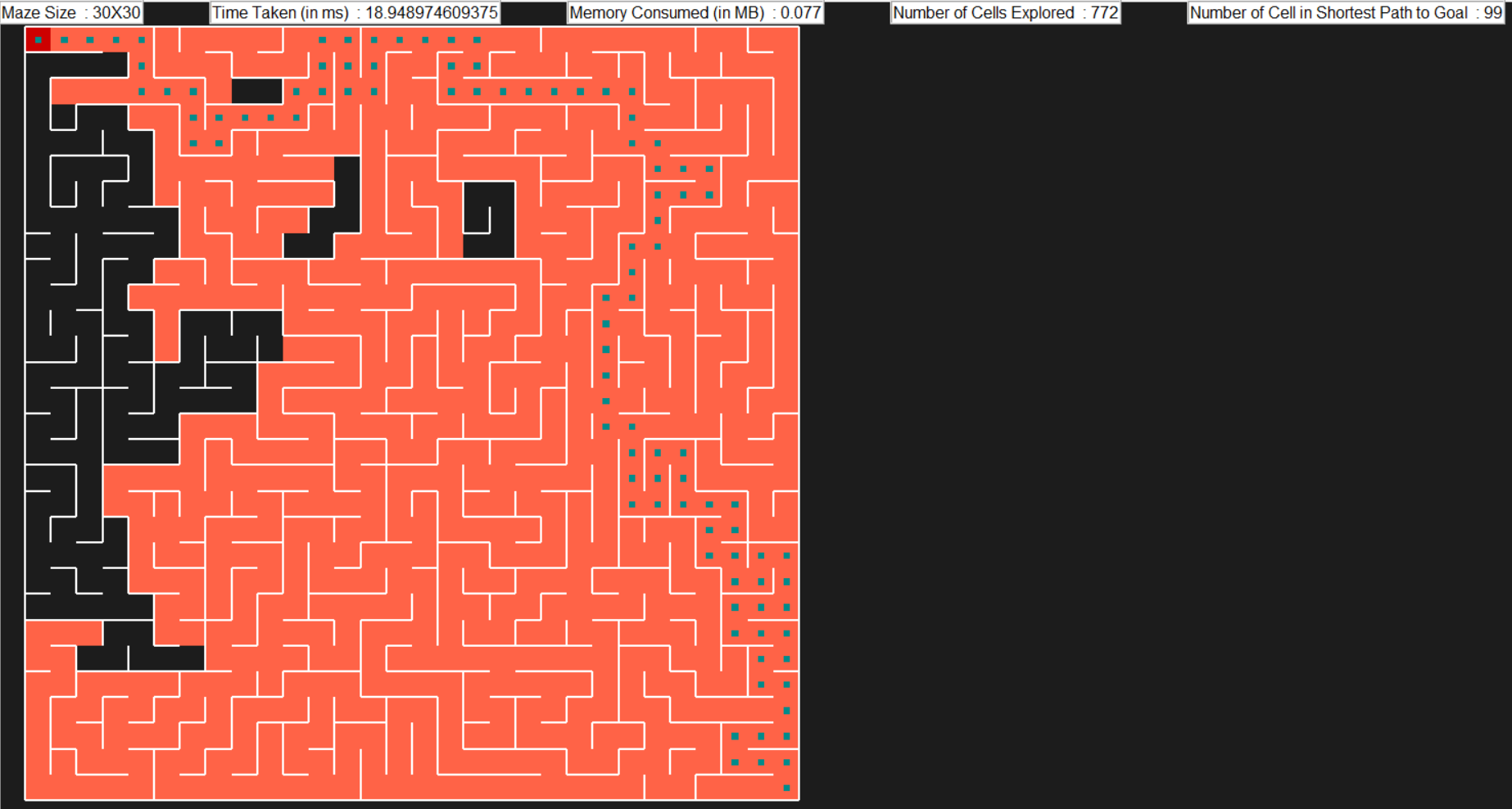
****

1. **A Star Search with Euclidian Distance as Heuristics**

****

****

****

****

****

****

1. **MDP Value Iteration**

****

****

****

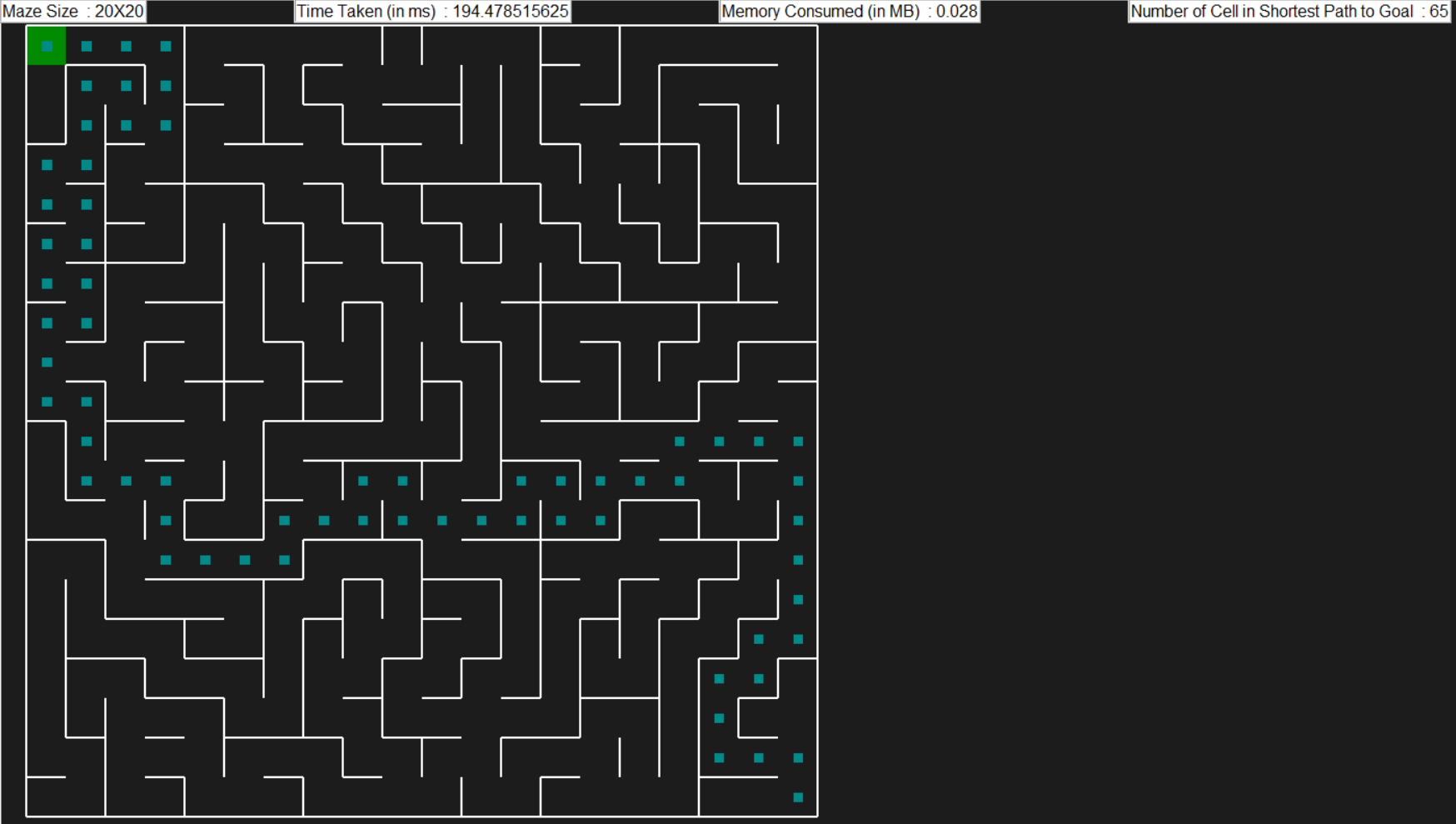
****

****

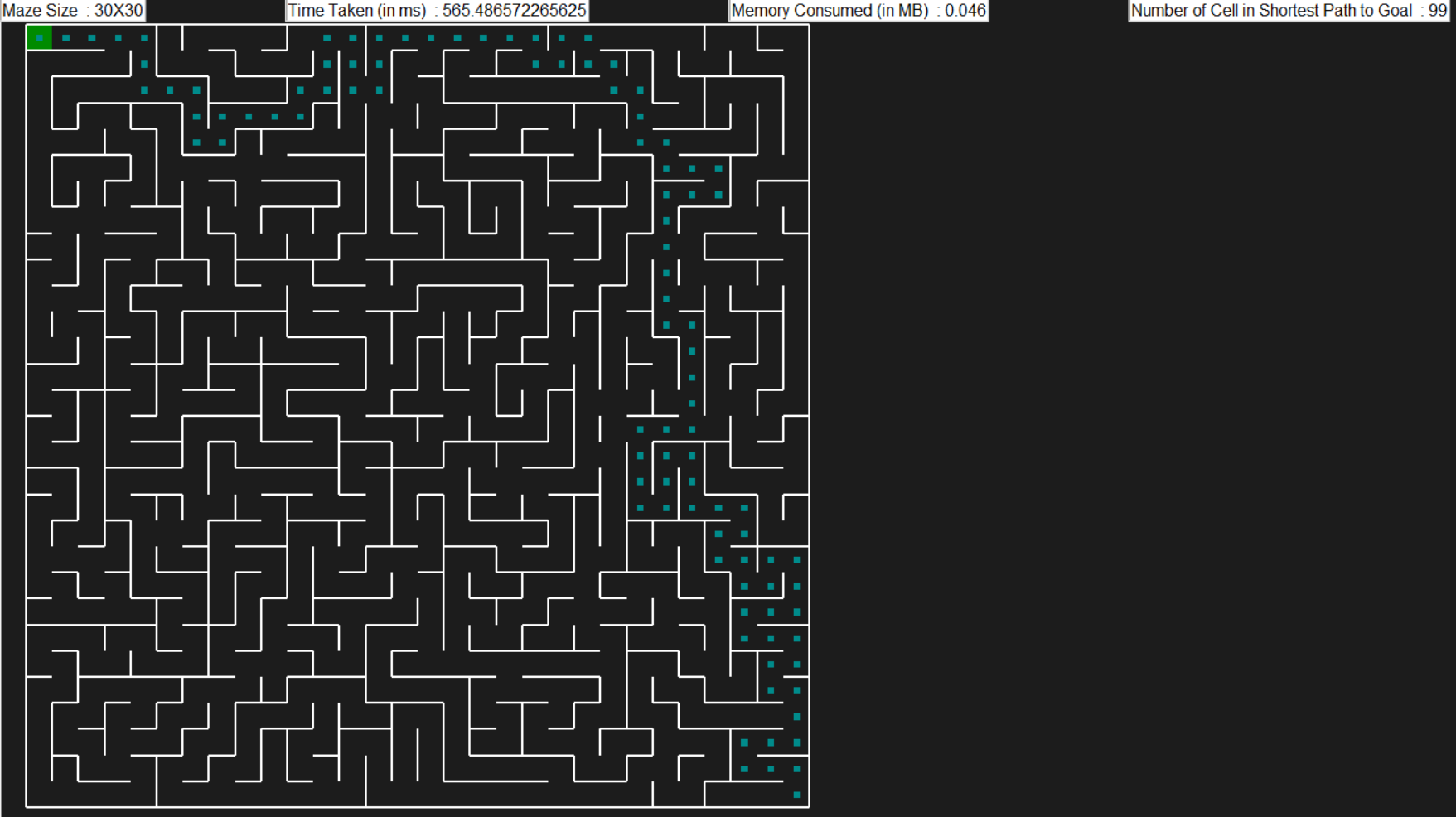
****

1. **MDP Policy Iteration**

****

****

****

****

****

****

**References**

1. https://github.com/MAN1986/pyamaze/tree/main/Demos
2. https://github.com/SparkShen02/MDP-with-Value-Iteration-and-Policy-Iteration
3. <https://www.youtube.com/playlist?list=PLWF9TXck7O_zsqnufs62t26_LJnLo4VRA>

**Code Execution Instructions**

1. Before running this code, please execute command **pip install pyamaze,** if this is not executed before
2. Unzip file code.zip
3. Please ensure following csv files are present in same directory as python notebooks

* Maze\_20X20.csv
* Maze\_30X30.csv
* Maze\_40X40.csv

1. Execute each of python notebooks provided in code folder implementing different maze search algorithms

**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)*

**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)*

**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)*

**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)*

**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)*

**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>

1. **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. | |
|  | | """ | |