

# CS7IS2: Artificial Intelligence Assignment 1

## Maze Solver

Kaiyu Chen  
23330889  
chenka@tcd.ie

School of Computer Science and Statistics  
Trinity College Dublin, The University of Dublin  
March 2024

### Table of Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Maze and GUI</b>	<b>2</b>
2.1	Maze . . . . .	2
2.2	GUI . . . . .	2
<b>3</b>	<b>Algorithms</b>	<b>2</b>
3.1	Graph Search Algorithms . . . . .	3
3.1.1	Breadth First Search . . . . .	3
3.1.2	Depth First Search . . . . .	4
3.1.3	A* Search . . . . .	4
3.2	Markov Decision Process . . . . .	4
3.2.1	Value Iteration . . . . .	4
3.2.2	Policy Iteration . . . . .	4
<b>4</b>	<b>Evaluation</b>	<b>5</b>
4.1	Path Steps, Time, and Memory Comparsion . . . . .	6
4.2	Search Path Steps Comparsion for Graph Search Algorithms . . . . .	8
4.3	Heat Map for MDP Algorithms . . . . .	8
<b>5</b>	<b>Conclusion</b>	<b>10</b>
<b>6</b>	<b>Solution of the Maze by All the Algorithms</b>	<b>10</b>
<b>A</b>	<b>Code of BFS</b>	<b>13</b>
<b>B</b>	<b>Code of DFS</b>	<b>14</b>
<b>C</b>	<b>Code of A*</b>	<b>15</b>
<b>D</b>	<b>Code of Value Iteration</b>	<b>17</b>
<b>E</b>	<b>Code of Policy Iteration</b>	<b>19</b>
<b>F</b>	<b>Origin Data</b>	<b>22</b>
<b>G</b>	<b>License of pyamaze</b>	<b>23</b>
<b>H</b>	<b>License of matplotlib</b>	<b>23</b>
<b>I</b>	<b>License of numpy</b>	<b>24</b>

# 1 Introduction

The main goal of assignment 1 is to create a maze in python and solve it using different algorithms. As mentioned in the assignment document, it is allowed to reuse existing open source maze generator. The module pyamaze is created for the easy generation of random maze and apply different search algorithm efficiently[?]. Thus, in this assignment, I have used the pyamaze module to create mazes and solve the mazes using different algorithms. The algorithms used are: Breadth First Search, Depth First Search, A\* Search, and Markov Decision Process, including the value iteration and policy iteration.

NumPy is the fundamental package for scientific computing in Python[?]. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python[?]. To visualize the performance of different algorithms, these two open source libraries are used in this assignment when visualize the performance.

All license of the used libraries and source code are included in the appendix.

The demonstration of this assignment can be found on YouTube(<https://youtu.be/wuz-iA9GuFo>).

The code of this assignment can be found on GitHub(<https://github.com/cky008/MazeSolver>).

## 2 Maze and GUI

### 2.1 Maze

As mentioned in the assignment document, the sizes of mazes for which to analyze the performance are not pre-specified. The maze is generated using the pyamaze module. However, the maze is generated randomly. To visualize the performance, it is better to compare the performance of different algorithms on the same maze. To achieve the goal of comparing the performance of different algorithms, I have created many mazes with different sizes and saved them in the `savedMaze` folder. Any .csv maze file generated by pyamaze and stored in the `savedMaze` folder can be automatically loaded to the program. The file can be selected by the user in the GUI as shown in Figure 1.

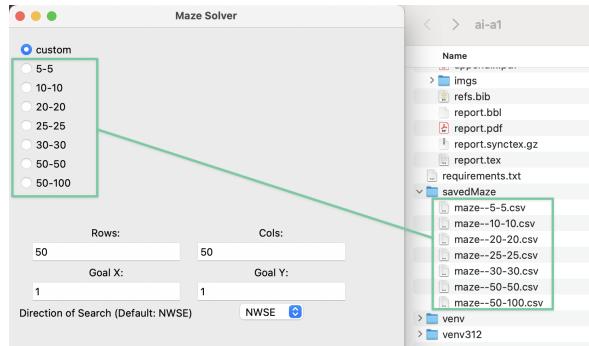


Figure 1: Auto Load Saved Maze From `savedMaze`

I have tried with different sizes of mazes, including: 5x5, 10x10, 20x20, 25x25 30x30, 40x40, 50x50, 75x75, 50x100. For some mazes that size over 75x75, some algorithms failed to get the correct final path for the maze. Since the goal is to analyze the performance of different algorithms generally, the mazes with size 5x5, 20x20, 30x30, 50x50, 50x100 are used to evaluate the performance of the algorithms in this assignment. Thus, those selected maze will be evaluated in the following sections, Figure 2 is the screenshot of those mazes.

### 2.2 GUI

I've programmed a GUI for ease of use. The GUI has the capability to show the maze saved in `savedMaze` folder as some checkboxes. Instead of choosing the saved maze, the user can also generate a random maze by specifying the size of the maze. For any type of maze, the user can choose the goal of the maze, the search direction of search algorithms, and the algorithm to solve the maze. For the three search algorithms, the user can choose to show the search path of the algorithms or not. For this option, the solver will show the search path firstly(then will disappear), then show the final path. When click run button, the algorithms selected will be used to calculate the path and the pyamaze will show the final path, and the performance of the algorithms will be shown in the GUI as label.

Besides, when select the algorithms, there will also be some figures shown from the matplotlib library to visualize the performance of the algorithms. The figure will show the path steps (including the start node and the goal node), time cost(in milliseconds), and the memory peak(in MB) for each selected algorithm. For graph search algorithms, there will also be a figure to show the search path steps of the maze. For the MDP algorithms, there will also be a figure to show the heat map of each node after the iteration. Those figure will be used for the evaluation later in this report. The figure will automatically saved to the `imgaes` folder as .eps file for the ease of use.

## 3 Algorithms

In this assignment, I've developed five different algorithms to solve the maze. For better reuse of the code, I've created classes for each algorithm. The algorithms are: Breadth First Search(BFS), Depth First Search(DFS), A\* Search, Value Iteration, and Policy Iteration. The BFS, DFS and A\* is based on the same parent class `GraphSearchAlgorithms` and the Value Iteration and Policy Iteration algorithms are based on the same parent class `MarkovDecisionProcess`. To use each class of algorithms, it is required to create an instance of the class with maze parameter `m` (import from pyamaze) and goal parameter `goal` (tuple).

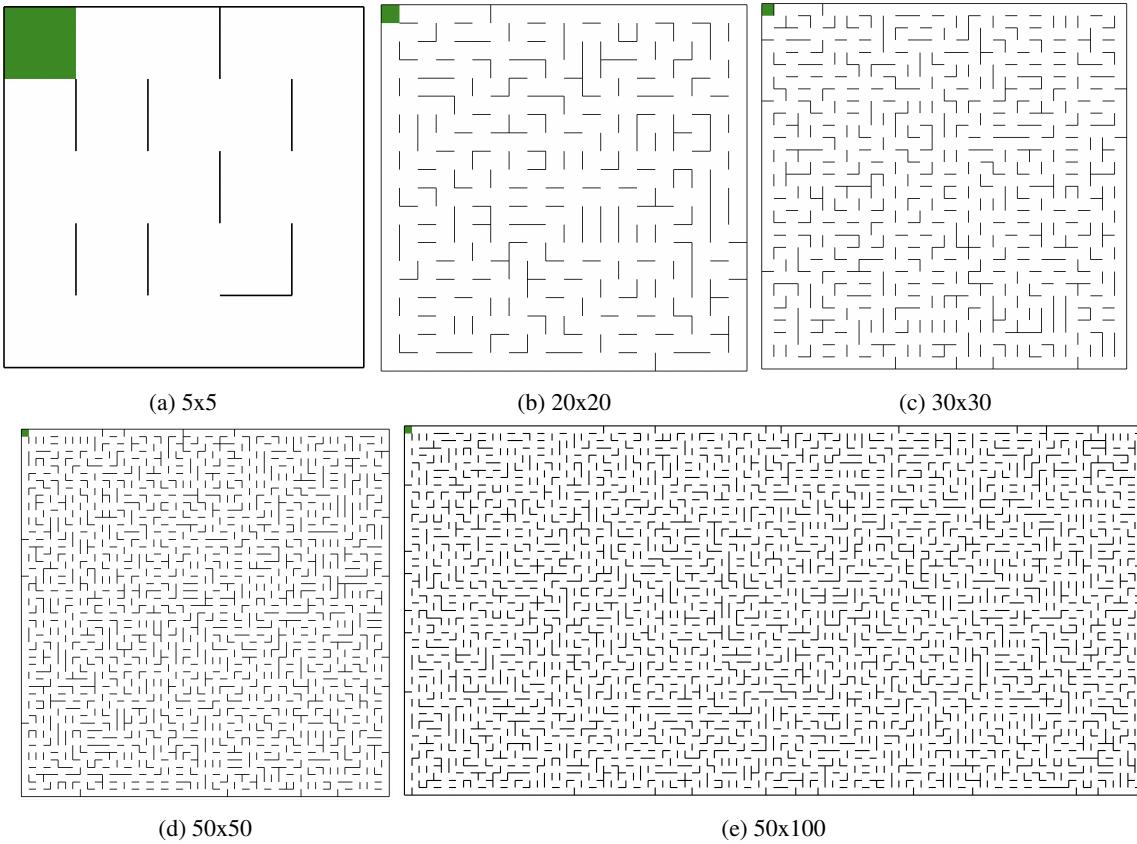


Figure 2: Used Maze

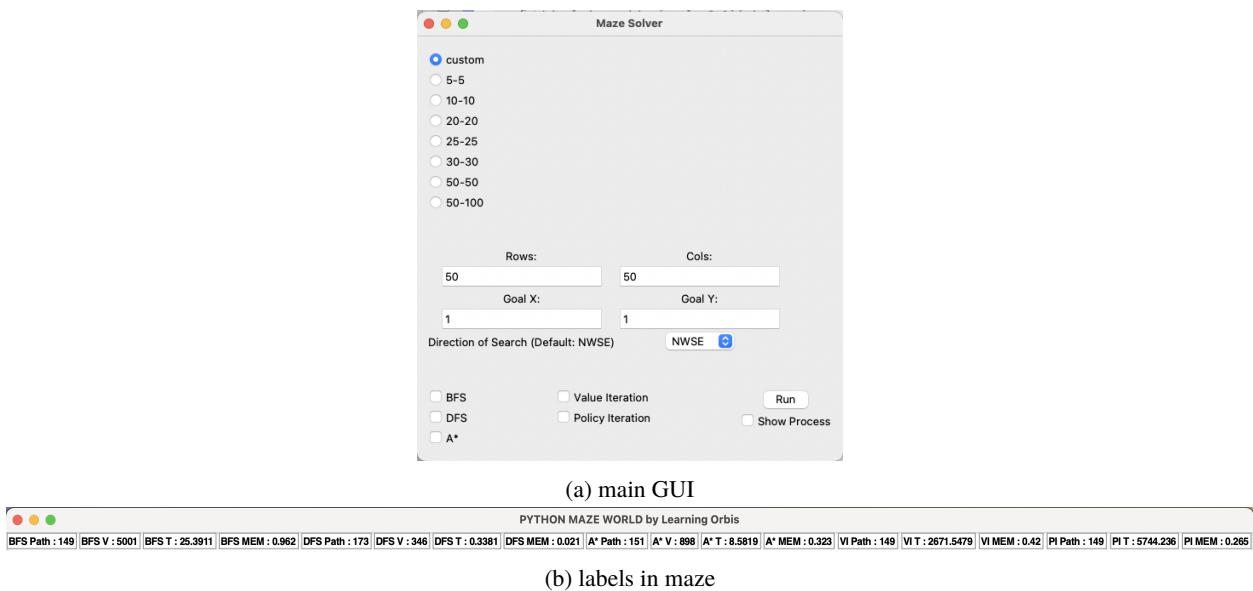


Figure 3: GUI of the program

### 3.1 Graph Search Algorithms

`GraphSearchAlgorithms` is the base class for the BFS, DFS and A\* algorithms. This class initializes some data structures and methods that are used by the three algorithms.

In this class, `start_memory_tracing` and `stop_memory_tracing` are used to trace and return the memory peak of the algorithm to `self.memory_peak`. `validate_node` is used to validate if the specific node is inside the maze and if it is visited. `compute_next_node` works with `validate_node` to calculate the next node of current node. `traceback_final_path` is used to trace back the dictionary `self.origin_dict` to get the final path of the maze. `execute_algorithm` is defined as an abstract method to be implemented by the child class. `get_final_path` is defined to get the final path, search path, cost time and memory peak of the algorithms.

### 3.1.1 Breadth First Search

BFS is the child class of `GraphSearchAlgorithms`. This class will be initialized with the variables and methods inherited from the parent class, and the start node will be put into queue.

`execute_algorithm` is the main method of this class. It execute the BFS algorithm and trace the time and memory for evaluation. This method uses queue to store all the pending nodes. This method works with `compute_next_node` to iterate each neighbors of current node. The final path is gotten from `traceback_final_path`.

### 3.1.2 Depth First Search

DFS is the child class of `GraphSearchAlgorithms`. This class will be initialized with the variables and methods inherited from the parent class. When initializing, `start_node` will be put into `final_path` and `search_path`.

`execute_algorithm` is the main method of this class. It execute the DFS algorithm and trace the time and memory for evaluation. Instead of using queue in BFS, this method uses `final_path` as a stack to store all the pending nodes. This method also works with `compute_next_node` to iterate each neighbors of current node. Also, the final path is gotten from `traceback_final_path`.

### 3.1.3 A\* Search

`AStar` is the child class of `GraphSearchAlgorithms`. This class will be initialized with the variables and methods inherited from the parent class. The `g_socre` is initialized with a 0 g score start node. The `priority_queue` is also initialized to used to manage nodes during the search based on their total cost.

The `h_score` method is used to calculate the heuristic cost of the node. It will check if the `manhattan_flag` is True or False. If it is True, the method will calculate the manhattan distance between the node and the goal node. If it is False, the method will calculate the euclidean distance between the node and the goal node. While Euclidean distance may overestimate and lead to A\* algorithm choosing a suboptimal path, Manhattan distance does not overestimate the actual distance to the target in a maze that can only move up, down[?] [?]. Thus, the manhattan distance is used in this assignment.

`execute_algorithm` is the main method of this class. It execute the A\* algorithm and trace the time and memory for evaluation. This algorithm is based on the `priority_queue`. This algorithm calculates the cost of the next node and compares it within a while loop Also, the final path is gotten from `traceback_final_path`.

## 3.2 Markov Decision Process

`MarkovDecisionProcess` is the the base class for the value iteration and policy iteration algorithms. This class initializes some data structures and methods that are used by the two algorithms. In this class, `start_memory_tracing` and `stop_memory_tracing` are used to trace and return the memory peak of the algorithm to `self.memory_peak`. `get_final_path` is defined to get the `final_path`, `cost_time` and `memory_peak` of the algorithms.

`discount_factor` and `convergence_threshold` are set to 0.9 and 0.000001 respectively in this assignment. Those two are important parameters for the MDP algorithms. I've tried with different values such as 0.1, 0.3, 0.5, 0.9 for discount factor, and found that 0.9 and 0.000001 are the best values for the performance of my MDP algorithms.

`get_direction_for_current_node` is used to get the direction of the current node when get the `final_path`. This method is passed to the child class for implementation since that two iteration are not same. `execute_algorithm` is defined as an abstract method to be implemented by the child class either. `plot_maze_weights` is used to draw the heat map of the maze after the iteration.

### 3.2.1 Value Iteration

`ValueIteration` is the child class of `MarkovDecisionProcess`. This class will be initialized with the variables and methods inherited from the parent class. Value iteration is initialized by setting the transition value to 10 for the goal node and 0 for all other nodes. Also, the transition reward is initialized to 100 for the goal node and 0 for all other nodes.

`get_direction_for_current_node` return data based on `max(current_node, key=current_node.get)` and `transition_dictionary`. `execute_iteration` is the main method of this class. This method updates the value of each node with constant iterations until the convergence condition is reached. The mathematical formulas that can be referred to are:

$$V(s) = \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)V(s') \right\}$$

Which is calculated by using code:

```
1 next_transition_value = self.transition_probability [ direction ] * ( self.
    transition_reward [current_node] + self.discount_factor * self.transition_value [
    next_node])
2 temp_transition_value.append( next_transition_value )
3 self.transition_dictionary [current_node][ direction ] = next_transition_value
4 best_transition_value = max(temp_transition_value)
```

### 3.2.2 Policy Iteration

`PolicyIteration` is the child class of `MarkovDecisionProcess`. This class will be initialized with the variables and methods inherited from the parent class. Policy Iteration algorithm is initialized by setting the transition value and reward to 1 for the goal node and 0 for all other nodes. The default policy is set to N (North/Up).

`get_direction_for_current_node` return data based on `policy` direction. is defined to conduct the policy evaluation based on the mathematical formula:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P(s'|s, \pi(s))V^\pi(s')$$

Which is calculated by using code:

```

1 temp_node_transition_value [current_node][ direction ] = self . transition_probability [ direction ] * ( self . transition_reward [current_node] + self . discount_factor * next_transition_value )
2 self . transition_dictionary [current_node][ current_policy ] = ( self . transition_probability [ current_policy ] * ( self . transition_reward [current_node] + next_transition_value * self . discount_factor ))
3 if abs( self . transition_value [current_node] - ( self . transition_probability [ current_policy ] * ( self . transition_reward [current_node] + next_transition_value * self . discount_factor ))) > self . convergence_threshold :
4     self . transition_value [current_node] = self . transition_probability [ current_policy ] *
        ( self . transition_reward [current_node] + next_transition_value * self .
        discount_factor )
5     value_converged_flag = False

```

`execute_iteration` is the main method of this class. It updates the value of each node by doing policy evaluation with `calculate_transition_value` and then performs the policy improvement. The policy improvement is based on the mathematical formula:

$$\pi'(s) = \arg \max_{a \in A} \left\{ R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s') \right\}$$

Which is calculated by using code:

```

1 for current_node in self.m.grid:
2     if current_node == self.goal:
3         continue
4     current_node_transition_value = self . calculate_transition_value (current_node)
5     current_node_transition_policy = get_direction ( current_node_transition_value [
        current_node ])
6     if current_node_transition_policy != self . policy [current_node]:
7         self . policy [ current_node ] = current_node_transition_policy
8         policy_converged_flag = False

```

Policy evaluation and policy improvement are alternated until the policy converges.

## 4 Evaluation

Table 1 summarizes the performance of the algorithms on the selected mazes. It can be discovered that the DFS algorithm has the best performance in terms of time and memory, however, the path length is the longest. This is because the DFS algorithm tends to explore a branch of the maze in depth until a dead end is reached or a goal is found, a strategy that is not always effective in finding the shortest path, but accomplishes the search task with low computational time and memory consumption. Also, sometimes the A\* algorithm has the best performance in terms of visited nodes. To have a better understanding of the performance, the following sections will use figure to analyze.

Maze Size	Algorithm	Path Length	Calculated Time (ms)	Memory Peak (MB)	Visited Nodes
5x5	BFS	9	0.2191	0.004	26
	DFS	<b>11</b>	<b>0.0348</b>	<b>0.001</b>	22
	A*	9	0.1507	0.005	<b>15</b>
	VAL	9	12.697	0.002	-
	POL	9	3.583	0.002	-
20x20	BFS	39	2.3408	0.061	401
	DFS	<b>45</b>	<b>0.1132</b>	<b>0.005</b>	90
	A*	39	0.4311	0.02	<b>49</b>
	VAL	39	214.5798	0.034	-
	POL	39	102.3591	0.022	-
30x30	BFS	59	5.2621	0.09	901
	DFS	<b>63</b>	<b>0.1237</b>	<b>0.007</b>	<b>126</b>
	A*	61	3.4239	0.101	326
	VAL	59	487.3381	0.076	-
	POL	59	334.6488	0.049	-
50x50	BFS	99	13.545	0.244	2501
	DFS	<b>123</b>	<b>0.2058</b>	<b>0.017</b>	255
	A*	99	2.1842	0.104	<b>243</b>
	VAL	99	1335.9559	0.21	-
	POL	99	1554.523	0.136	-
50x100	BFS	149	24.23	0.962	5001
	DFS	<b>173</b>	<b>0.3977</b>	<b>0.021</b>	<b>346</b>
	A*	151	7.443	0.323	898
	VAL	149	2660.815	0.42	-
	POL	149	6087.6012	0.265	-

Table 1: Maze Solving Algorithms Performance

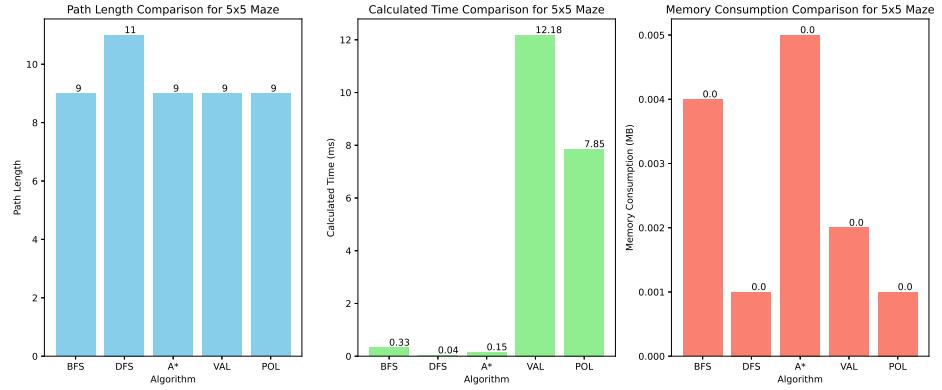
#### 4.1 Path Steps, Time, and Memory Comparsion

Figure 4 shows the path steps, time, and memory comparsion of the algorithms on the selected mazes. It can be found that A\* algorithm is also efficient comparing to the other algorithms, especially when the size of maze is getting bigger and bigger in terms of time. Overall, the efficiency of the graph search algorithms is better than the MDP algorithms. It is because that the MDP is based on the iteration, and the iteration is time-consuming.

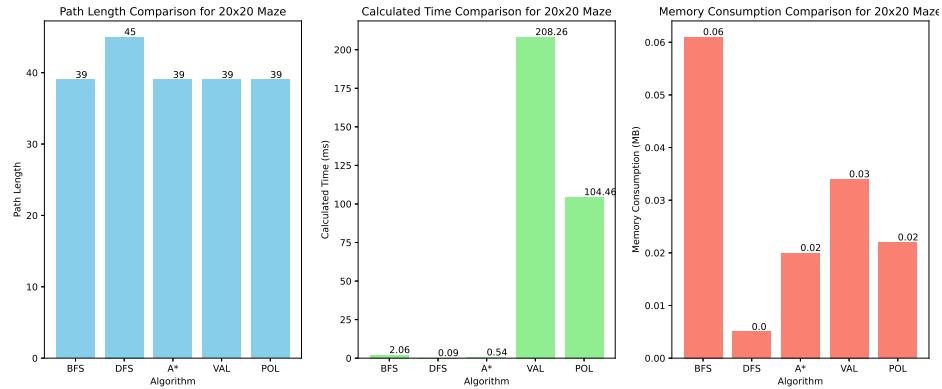
Comparing in MDP, the Policy Iteration is more efficient than the Value Iteration in terms of memory. When considering the time, the Value Iteration is more efficient than the Policy Iteration when the maze is smaller than 50x50. When the maze is bigger than 50x50, the Value Iteration is more efficient than the Policy Iteration in terms of time.

BFS is even less efficient than MDP mostly in terms of memory. This is probably because it iterates over all the nodes and the maze is not weighted.

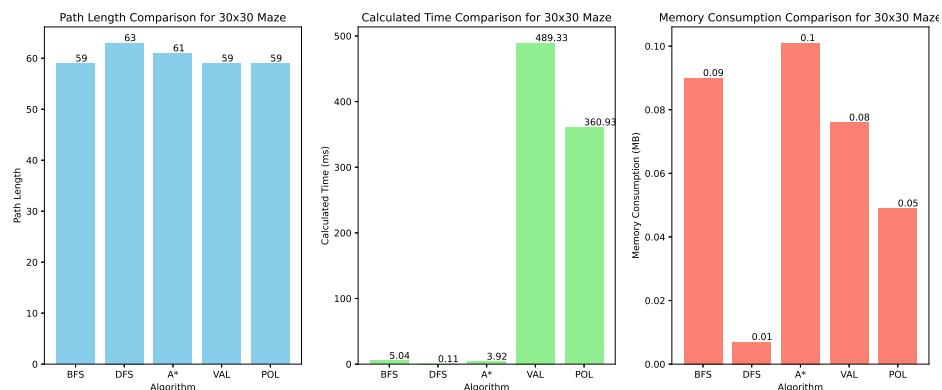
For mazes with multiple solutions, BFS, A\*, Value Iteration and Policy Iteration have the shortest paths. DFS has the longest path. This is the expected result.



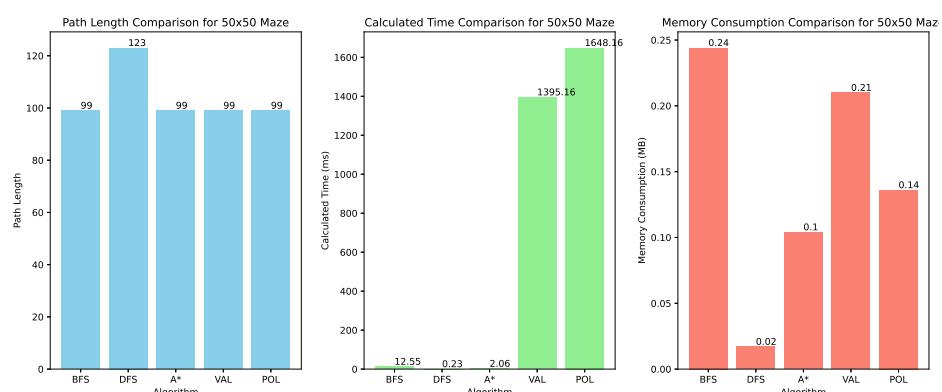
(a) 5x5



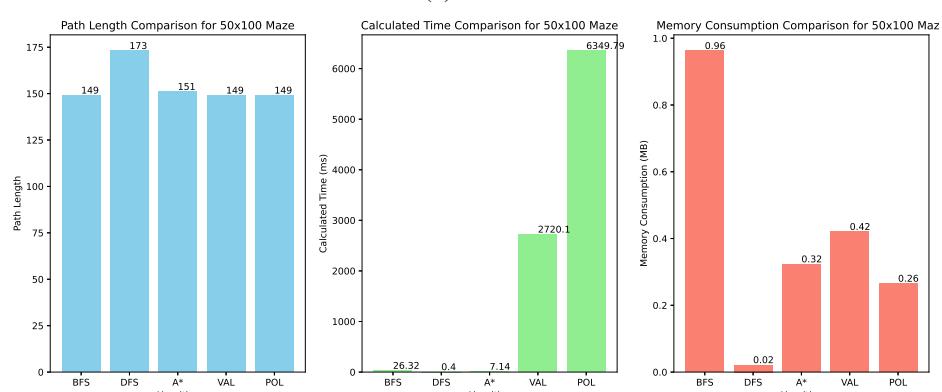
(b) 20x20



(c) 30x30



(d) 50x50



(e) 50x100

Figure 4: Path Steps, Time, and Memory Comparsion

## 4.2 Search Path Steps Comparsion for Graph Search Algorithms

Figure 5 shows the search path steps comparsion of the graph search algorithms on the selected mazes.

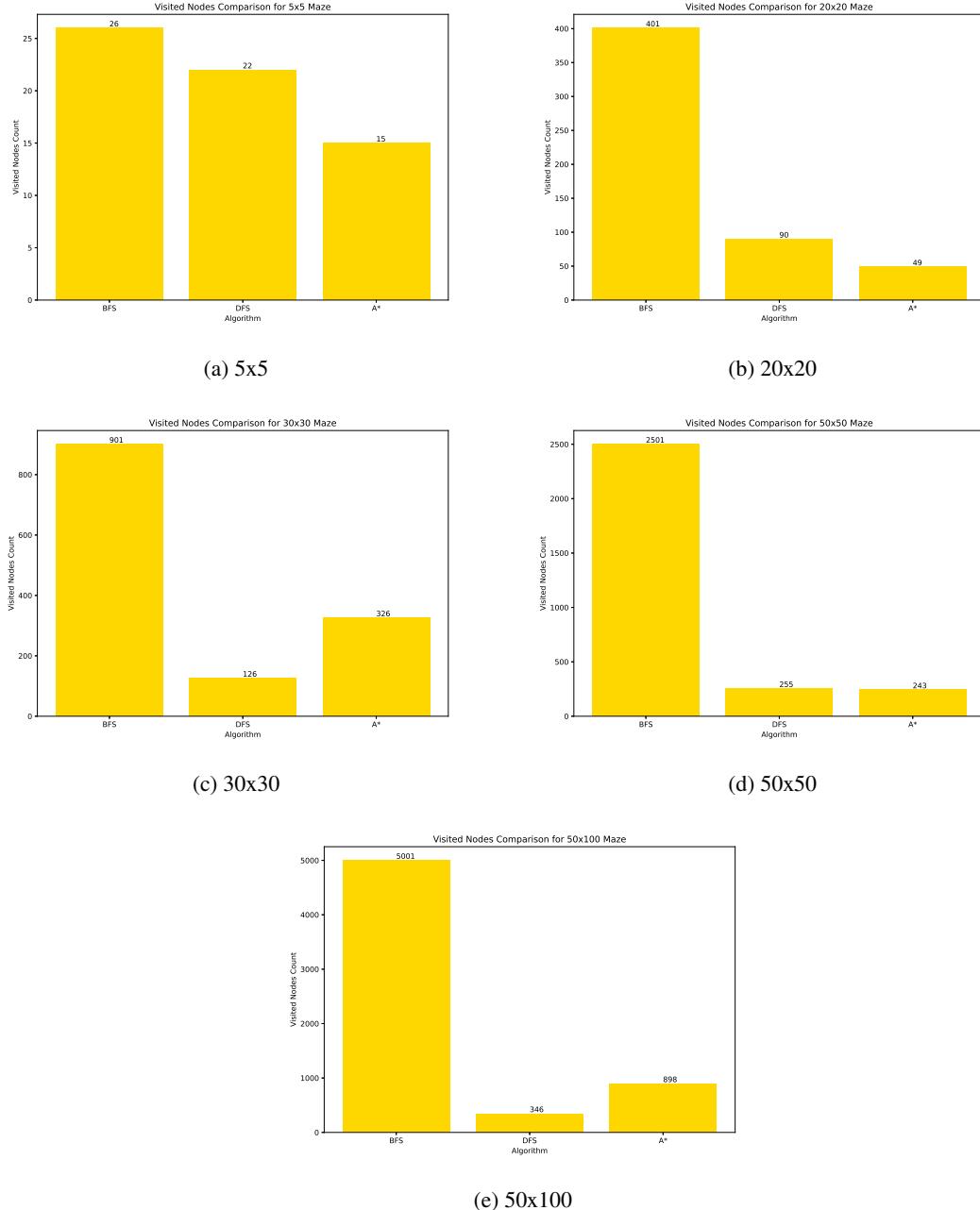


Figure 5: Search Steps Comparsion

In most cases, A\* and DFS explore the fewest nodes, BFS explores the most nodes.

## 4.3 Heat Map for MDP Algorithms

Figure 6 shows the heat map of the maze from MDP algorithms. This section is for visualizations.

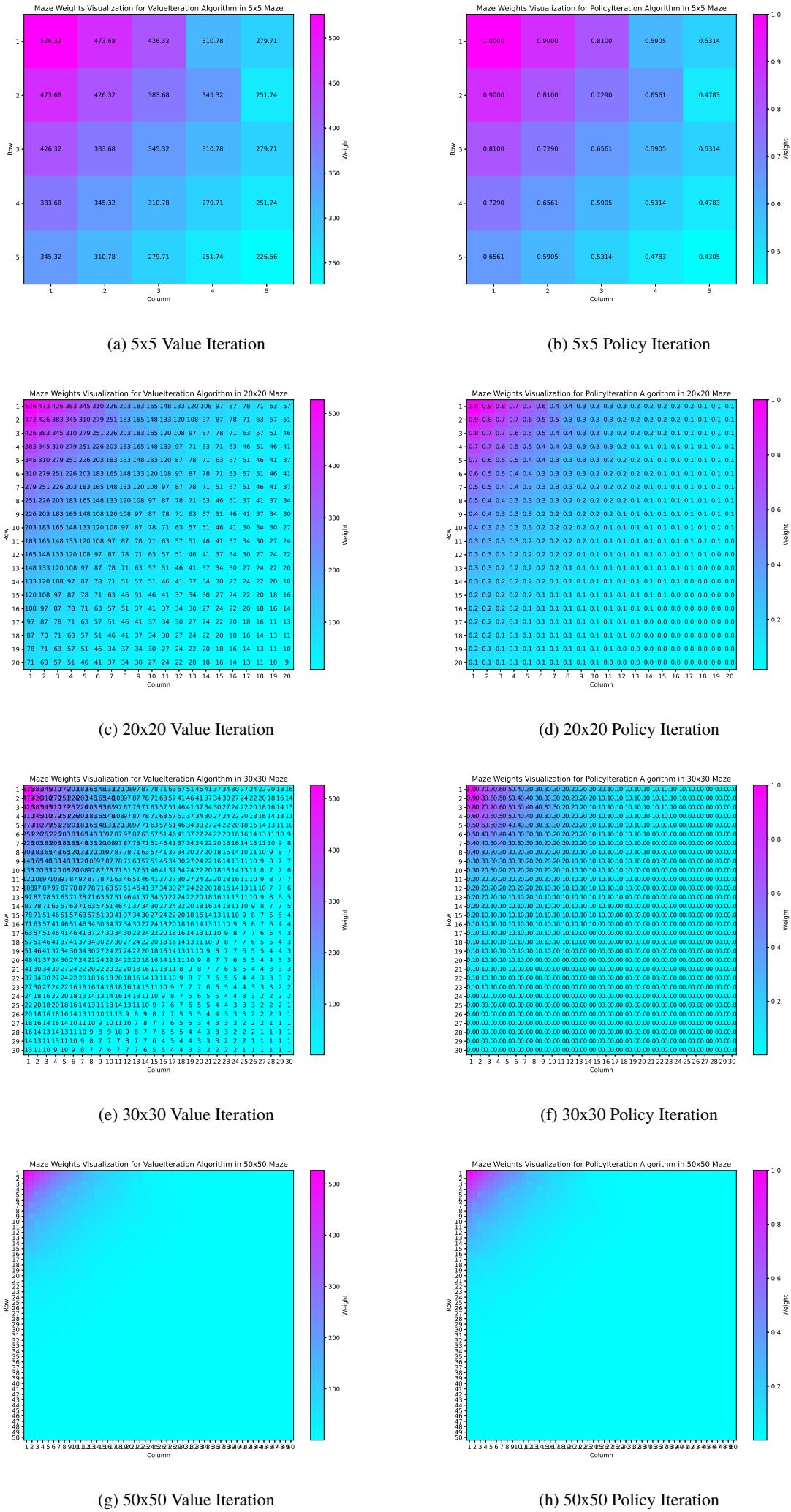
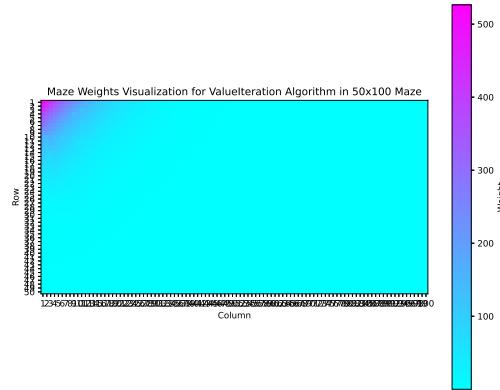
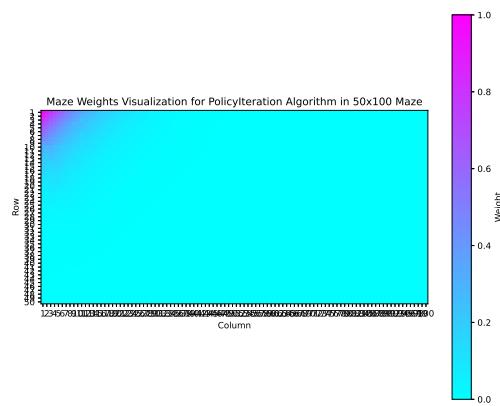


Figure 6: Heat Map of Markov Decision Process(1)



(a) 50x100 Value Iteration



(b) 50x100 Policy Iteration

Figure 7: Heat Map of Markov Decision Process(2)

## 5 Conclusion

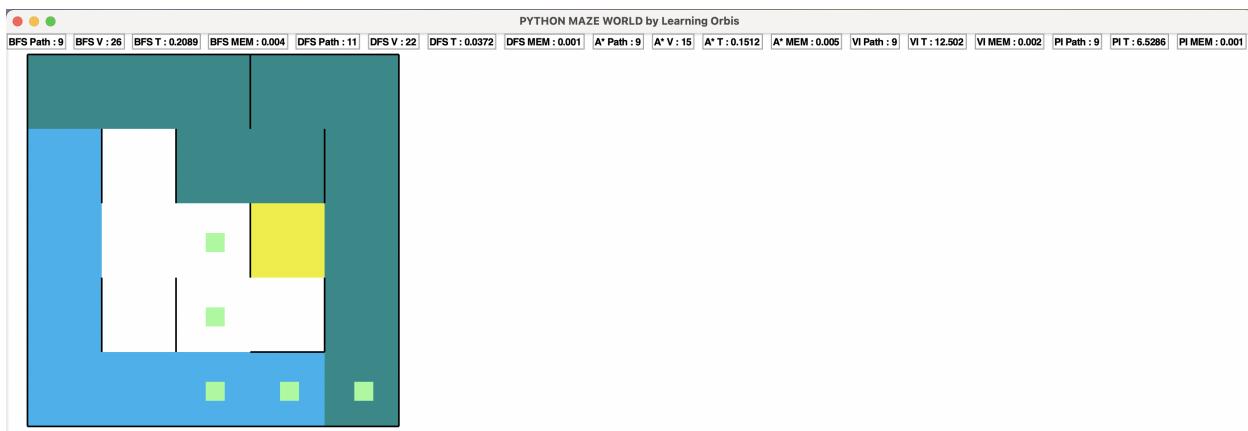
It can be concluded that it's important to consider which is the best algorithm in different situations. For example, if the maze is small and the goal is to find the shortest path, A\* and BFS is the best choice. In other words, before using an algorithm, it is necessary to analyze the requirements of the problem and determine the best solution. Overall, Graph Search Algorithms are less time-consuming and Markov Decision Processes are more time-consuming. However, the Markov decision process consumes less memory when solving for the best goal. DFS is the best choice if want to solve problems quickly and reduce memory and time overhead.

Overall, I got a lot from this assignment. Not only did I review the algorithms in detail, but I also reviewed OOP.

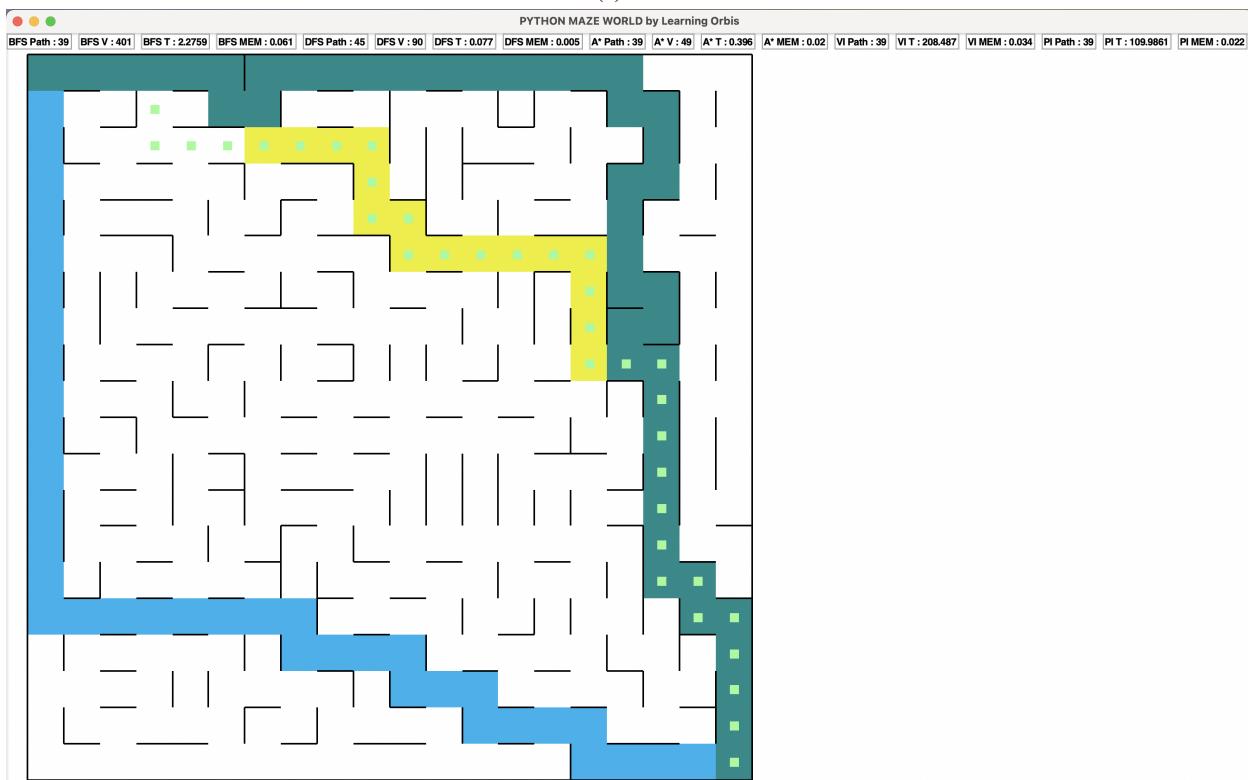
## 6 Solution of the Maze by All the Algorithms

This section is for showing the solutions of the selected maze by all the algorithms.

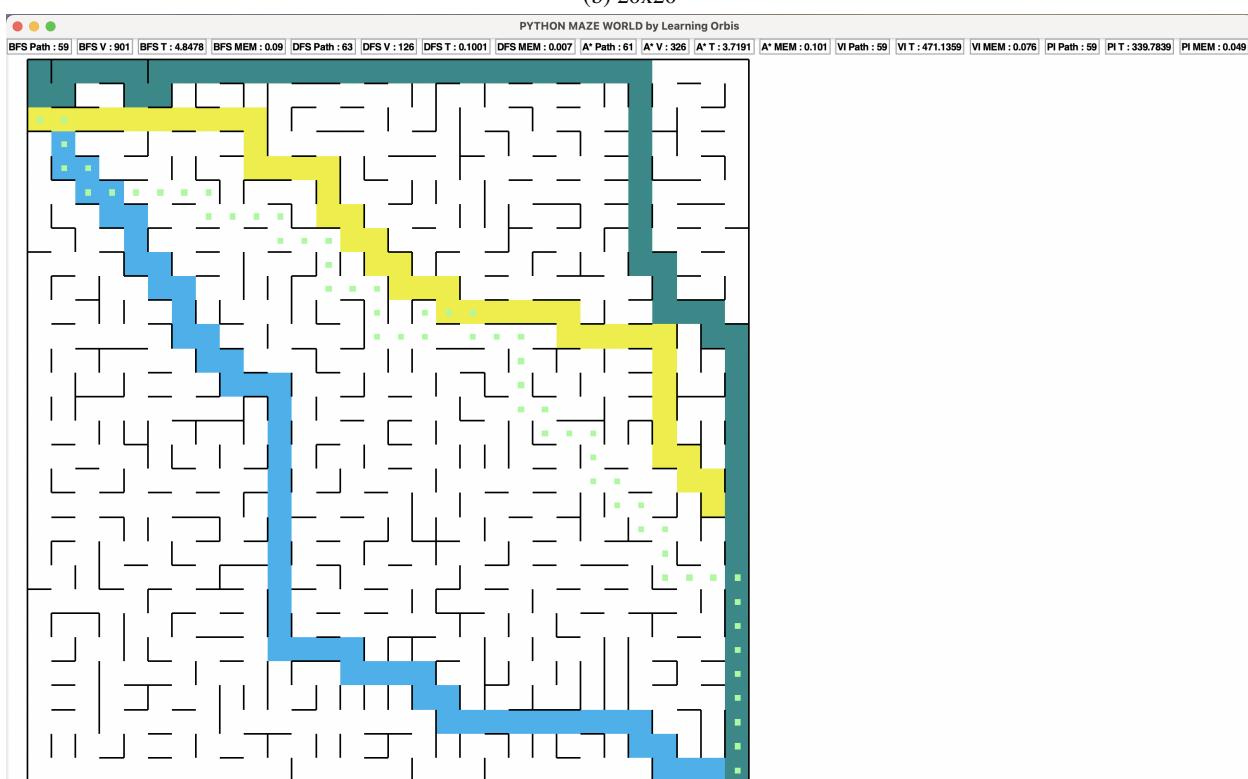
- The path with arrow is the explored/searched path, and the path with line or dots is the final path.
- Yellow color represents the final path of the algorithm BFS.
- Cyan color (dark green) represents the final path of the algorithm DFS.
- Green color (light green) represents the final path of the algorithm A\*.
- Blue color represents the final path of the algorithm Value Iteration.
- Red color represents the final path of the algorithm Policy Iteration.



(a) 5x5

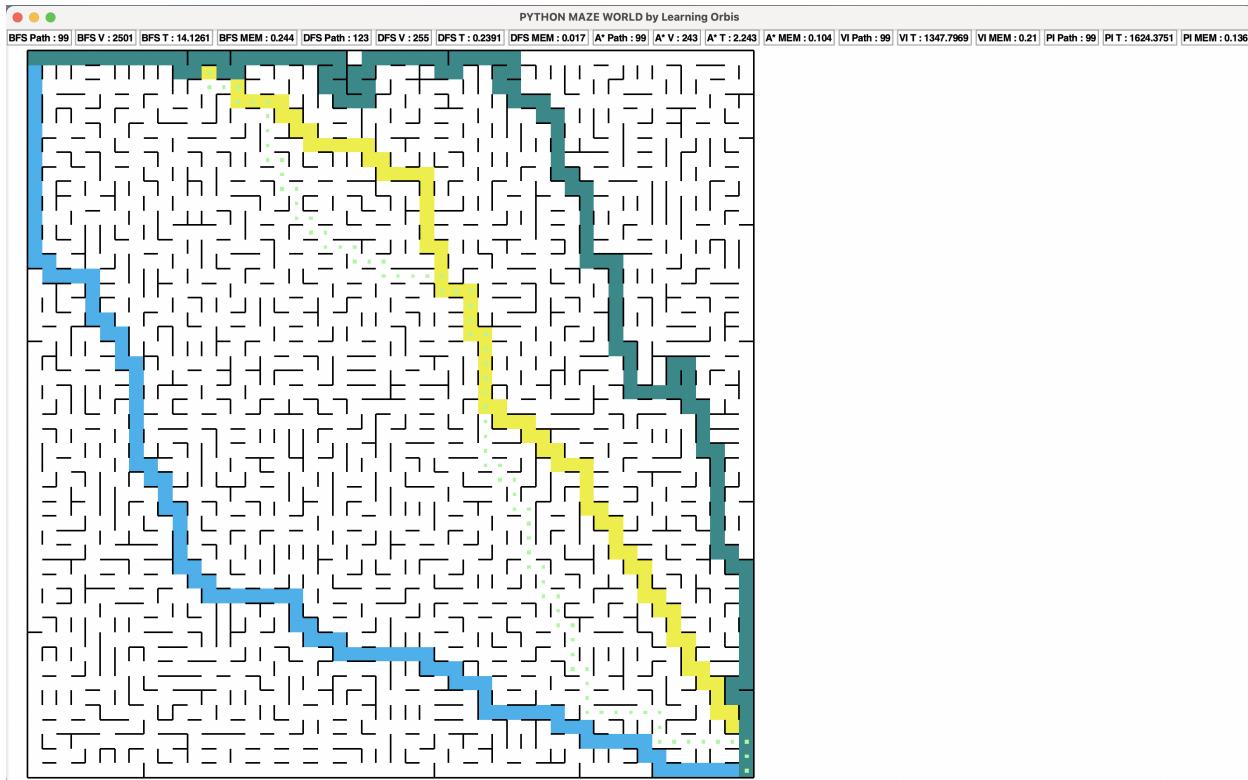


(b) 20x20

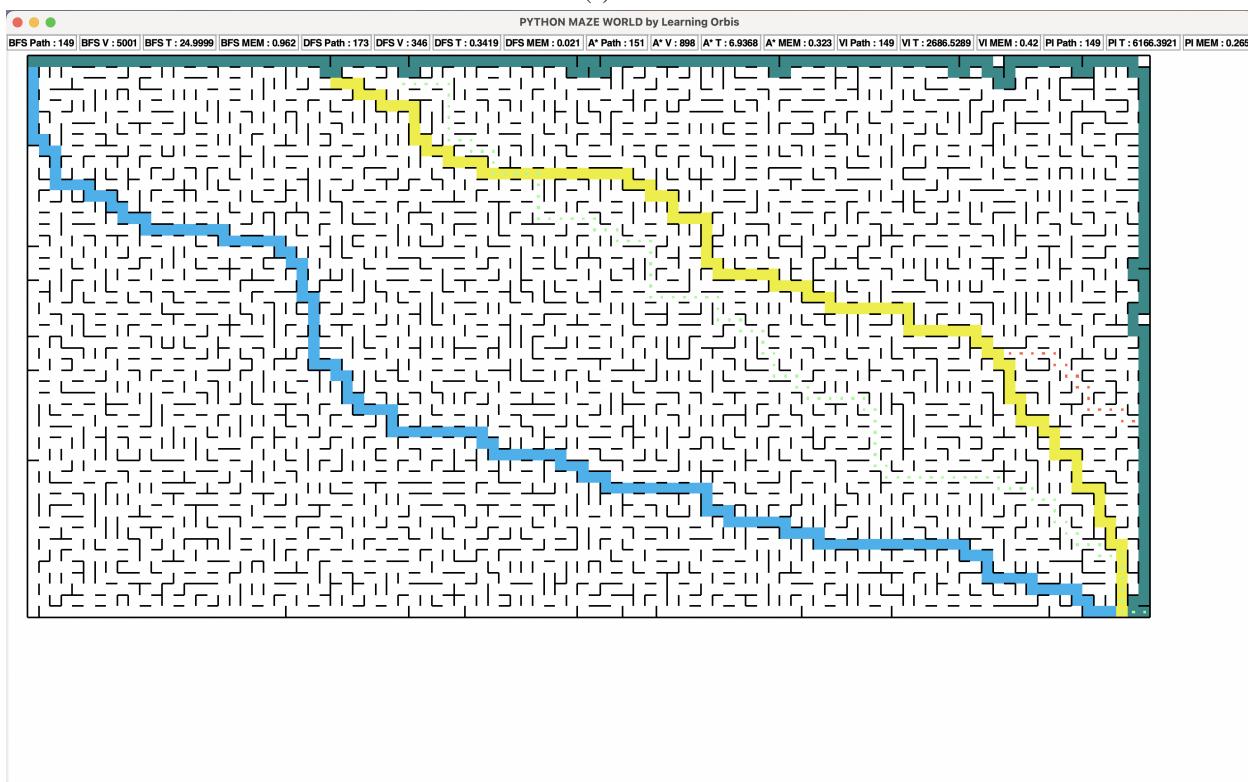


(c) 30x30

Figure 8: Solutions of the Maze by All the Algorithms(1)



(a) 50x50



(b) 50x100

Figure 9: Solutions of the Maze by All the Algorithms(2)

## A Code of BFS

```
1 from queue import Queue, PriorityQueue
2 import time
3 import tracemalloc as memory_trace
4 from math import sqrt
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 def start_memory_tracing():
9     memory_trace.stop()
10    memory_trace.start()
11
12 class GraphSearchAlgorithms:
13     def __init__(self, m, goal):
14         self.m = m
15         self.cost_time = 0
16         self.memory_peak = 0
17         self.goal = goal
18         self.start_node = (self.m.rows, self.m.cols)
19         self.visited = set()
20         self.nodes = 'NWE'
21         self.final_path = []
22         self.search_path = []
23         self.origin_dict = {}
24
25     def stop_memory_tracing(self):
26         _, self.memory_peak = memory_trace.get_traced_memory()
27         return self.memory_peak
28
29     def validate_node(self, current_node):
30         if current_node[0] < 1 or current_node[0] > self.m.rows or current_node[1] < 1
31             or current_node[1] > self.m.cols:
32             return False
33         if current_node in self.visited:
34             return False
35         return True
36
37     def compute_next_node(self, current_node):
38         for direction in self.nodes:
39             if self.m.maze_map[current_node][direction]:
40                 next_node = None
41                 if direction == 'N':
42                     next_node = (current_node[0] - 1, current_node[1])
43                 elif direction == 'W':
44                     next_node = (current_node[0], current_node[1] - 1)
45                 elif direction == 'S':
46                     next_node = (current_node[0] + 1, current_node[1])
47                 elif direction == 'E':
48                     next_node = (current_node[0], current_node[1] + 1)
49                 if self.validate_node(next_node):
50                     return next_node
51
52     def traceback_final_path(self):
53         final_path = []
54         current_node = self.goal
55         while current_node != self.start_node:
56             final_path.append(current_node)
57             current_node = self.origin_dict[current_node]
58         final_path.append(self.start_node)
59         return final_path[::-1]
60
61     def execute_algorithm(self):
62         pass
63
64     def get_final_path(self):
65         return self.final_path, len(self.search_path), self.cost_time, self.
66         memory_peak
67
68 class BFS(GraphSearchAlgorithms):
69     def __init__(self, m, goal):
70         super().__init__(m, goal)
```

```

69     self.queue = Queue()
70     self.queue.put( self.start_node )
71
72     def execute_algorithm( self ):
73         start = time.time()
74         start_memory_tracing()
75         while self.queue:
76             current_node = self.queue.get()
77             self.search_path.append(current_node)
78             if current_node == self.goal:
79                 self.final_path = self.traceback_final_path()
80                 end = time.time()
81                 self.stop_memory_tracing()
82                 self.cost_time = end - start
83                 return self.search_path, self.final_path
84             while True:
85                 new_position = self.compute_next_node(current_node)
86                 if new_position:
87                     self.queue.put(new_position)
88                     self.visited.add(new_position)
89                     self.origin_dict[new_position] = current_node
90                 else:
91                     break
92             return self.search_path, None

```

## B Code of DFS

```

1 from queue import Queue, PriorityQueue
2 import time
3 import tracemalloc as memory_trace
4 from math import sqrt
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 def start_memory_tracing():
9     memory_trace.stop()
10    memory_trace.start()
11
12 class GraphSearchAlgorithms:
13     def __init__(self, m, goal):
14         self.m = m
15         self.cost_time = 0
16         self.memory_peak = 0
17         self.goal = goal
18         self.start_node = (self.m.rows, self.m.cols)
19         self.visited = set()
20         self.nodes = 'NWSE'
21         self.final_path = []
22         self.search_path = []
23         self.origin_dict = {}
24
25     def stop_memory_tracing(self):
26         _, self.memory_peak = memory_trace.get_traced_memory()
27         return self.memory_peak
28
29     def validate_node(self, current_node):
30         if current_node[0] < 1 or current_node[0] > self.m.rows or current_node[1] < 1 or
31             current_node[1] > self.m.cols:
32             return False
33         if current_node in self.visited:
34             return False
35         return True
36
37     def compute_next_node(self, current_node):
38         for direction in self.nodes:
39             if self.m.maze_map[current_node][direction]:
40                 next_node = None
41                 if direction == 'N':
42                     next_node = (current_node[0] - 1, current_node[1])
43                 elif direction == 'W':
44

```

```

43     next_node = (current_node[0], current_node[1] - 1)
44     elif direction == 'S':
45         next_node = (current_node[0] + 1, current_node[1])
46     elif direction == 'E':
47         next_node = (current_node[0], current_node[1] + 1)
48     if self.validate_node(next_node):
49         return next_node
50
51 def traceback_final_path(self):
52     final_path = []
53     current_node = self.goal
54     while current_node != self.start_node:
55         final_path.append(current_node)
56         current_node = self.origin_dict[current_node]
57     final_path.append(self.start_node)
58     return final_path[::-1]
59
60 def execute_algorithm(self):
61     pass
62
63 def get_final_path(self):
64     return self.final_path, len(self.search_path), self.cost_time, self.
65     memory_peak
66
67 class DFS(GraphSearchAlgorithms):
68     def __init__(self, m, goal):
69         super().__init__(m, goal)
70         self.final_path = [self.start_node]
71         self.search_path = [self.start_node]
72
73     def execute_algorithm(self):
74         start = time.time()
75         start_memory_tracing()
76         while self.final_path:
77             current_position = self.final_path.pop()
78             self.search_path.append(current_position)
79             if current_position == self.goal:
80                 end = time.time()
81                 self.stop_memory_tracing()
82                 self.cost_time = end - start
83                 self.final_path.append(current_position)
84                 return self.search_path, self.final_path
85             self.visited.add(current_position)
86             new_position = self.compute_next_node(current_position)
87             if new_position:
88                 self.final_path.append(current_position)
89                 self.final_path.append(new_position)
90                 self.search_path.append(new_position)
91         return self.search_path, None

```

## C Code of A\*

```

1 from queue import Queue, PriorityQueue
2 import time
3 import tracemalloc as memory_trace
4 from math import sqrt
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 def start_memory_tracing():
9     memory_trace.stop()
10    memory_trace.start()
11
12 class GraphSearchAlgorithms:
13     def __init__(self, m, goal):
14         self.m = m
15         self.cost_time = 0
16         self.memory_peak = 0
17         self.goal = goal
18         self.start_node = (self.m.rows, self.m.cols)

```

```

19     self . visited = set ()
20     self . nodes = 'NWSE'
21     self . final_path = []
22     self . search_path = []
23     self . origin_dict = {}
24
25     def stop_memory_tracing(self):
26         _, self .memory_peak = memory_trace.get_traced_memory()
27         return self .memory_peak
28
29     def validate_node ( self , current_node):
30         if current_node [0] < 1 or current_node [0] > self .m.rows or current_node [1] < 1
31             or current_node [1] > self .m.cols:
32                 return False
33         if current_node in self . visited :
34             return False
35         return True
36
37     def compute_next_node(self, current_node):
38         for direction in self .nodes:
39             if self .m.maze_map[current_node][direction]:
40                 next_node = None
41                 if direction == 'N':
42                     next_node = (current_node [0] - 1, current_node [1])
43                 elif direction == 'W':
44                     next_node = (current_node [0], current_node [1] - 1)
45                 elif direction == 'S':
46                     next_node = (current_node [0] + 1, current_node [1])
47                 elif direction == 'E':
48                     next_node = (current_node [0], current_node [1] + 1)
49                 if self . validate_node (next_node):
50                     return next_node
51
52     def traceback_final_path ( self ):
53         final_path = []
54         current_node = self .goal
55         while current_node != self . start_node :
56             final_path.append(current_node)
57             current_node = self . origin_dict [current_node]
58         final_path.append( self . start_node )
59         return final_path [:-1]
60
61     def execute_algorithm ( self ):
62         pass
63
64     def get_final_path ( self ):
65         return self . final_path , len( self . search_path ), self . cost_time , self .
66             memory_peak
67
68     class AStar(GraphSearchAlgorithms):
69         def __init__ ( self , m, goal):
70             super().__init__(m, goal)
71             self .g_score = { self . start_node : 0}
72             self .priority_queue = PriorityQueue()
73             self .manhattan_flag = True
74
75             def h_score( self , node, manhattan=True):
76                 if manhattan:
77                     return abs(node[0] - self .goal [0]) + abs(node[1] - self .goal [1])
78                 else: # Euclidean
79                     return sqrt (((node[0] - self .goal [0]) ** 2) + ((node[1] - self .goal [1])
80                         ** 2))
81
82             def execute_algorithm ( self ):
83                 start = time.time()
84                 start_memory_tracing()
85                 self .priority_queue.put(( self .g_score[ self . start_node ] + self .h_score( self .
86                     start_node ), self . start_node ))
87
88                 while self . priority_queue :
89                     _, current_position = self . priority_queue .get()
90                     self . search_path.append( current_position )

```

```

87     if current_position == self.goal:
88         end = time.time()
89         self.stop_memory_tracing()
90         self.cost_time = end - start
91         self.final_path = self.traceback_final_path()
92         return self.search_path, self.final_path
93     self.visited.add(current_position)
94     while True:
95         new_position = self.compute_next_node(current_position)
96         if new_position:
97             self.g_score[new_position] = self.g_score[current_position] + 1
98             self.priority_queue.put((self.g_score[new_position] + self.
99                                     h_score(new_position), new_position))
100            self.origin_dict[new_position] = current_position
101            self.visited.add(new_position)
102        else:
103            break
104    return self.search_path, None

```

## D Code of Value Iteration

```

1 from queue import Queue, PriorityQueue
2 import time
3 import tracemalloc as memory_trace
4 from math import sqrt
5 import numpy as np
6 import matplotlib.pyplot as plt
7 def start_memory_tracing():
8     memory_trace.stop()
9     memory_trace.start()
10
11 def get_direction(current_node):
12     return max(current_node, key=current_node.get)
13
14 class MarkovDecisionProcess:
15     def __init__(self, m=None, goal=None):
16         self.m = m
17         self.cost_time = 0
18         self.memory_peak = 0
19         self.final_path = {}
20         self.m = m
21         if goal is None:
22             raise AssertionError("Goal Cannot Be None")
23         self.goal = goal
24         self.start_node = (self.m.rows, self.m.cols)
25
26         self.discount_factor = 0.9
27         self.convergence_threshold = 0.000001
28         self.transition_probability = {'N': 1, 'S': 1, 'W': 1, 'E': 1}
29         self.transition_dictionary = {key: {subkey: 0 for subkey in self.m.maze_map[
30             key]} for key in self.m.maze_map}
31
32     def start_memory_tracing(self):
33         memory_trace.stop()
34         memory_trace.start()
35
36     def stop_memory_tracing(self):
37         _, self.memory_peak = memory_trace.get_traced_memory()
38         return self.memory_peak
39
40     def get_final_path(self):
41         next_node = [self.start_node]
42
43         while len(next_node) > 0:
44             current_node = next_node.pop()
45             if current_node == self.goal:
46                 break
47
48             direction = self.get_direction_for_current_node(current_node)

```

```

49         if direction == 'N':
50             _temp_next_node_ = (current_node[0] - 1, current_node[1])
51         elif direction == 'S':
52             _temp_next_node_ = (current_node[0] + 1, current_node[1])
53         elif direction == 'W':
54             _temp_next_node_ = (current_node[0], current_node[1] - 1)
55         elif direction == 'E':
56             _temp_next_node_ = (current_node[0], current_node[1] + 1)
57         else:
58             raise AssertionError("Invalid Direction")
59         self.final_path[current_node] = _temp_next_node_
60         next_node.append(_temp_next_node_)
61
62     return self.final_path, self.cost_time, self.memory_peak
63
64 def get_direction_for_current_node(self, current_node):
65     pass
66
67 def execute_iteration(self):
68     pass
69
70 def plot_maze_weights(self):
71     weights = np.zeros((self.m.rows, self.m.cols))
72     data = self.transition_value
73     for node, value in data.items():
74         weights[node[0] - 1, node[1] - 1] = value
75
76     plt.figure(figsize=(10, 8))
77
78     if self.m.rows <= 30 and self.m.cols <= 30:
79         for i in range(self.m.rows):
80             for j in range(self.m.cols):
81                 if self.__class__.__name__ == 'ValueIteration':
82                     if self.m.rows > 20 or self.m.cols > 20:
83                         plt.text(j, i, str(int(weights[i, j])), ha='center', va='center', color='black')
84                     else:
85                         plt.text(j, i, format(weights[i, j], '.2f'), ha='center', va='center', color='black')
86                 else:
87                     if self.m.rows > 20 or self.m.cols > 20:
88                         plt.text(j, i, format(weights[i, j], '.1f'), ha='center', va='center', color='black')
89                     else:
90                         plt.text(j, i, format(weights[i, j], '.4f'), ha='center', va='center', color='black')
91
92     plt.imshow(weights, cmap='cool', interpolation='nearest', origin='upper')
93     plt.colorbar(label='Weight')
94     plt.xticks(np.arange(self.m.cols), np.arange(1, self.m.cols + 1))
95     plt.yticks(np.arange(self.m.rows), np.arange(1, self.m.rows + 1))
96     plt.title(f'Maze_Weights_Visualization_{self.__class__.__name__}_{Algorithm_in_{self.m.rows}x{self.m.cols}_Maze}')
97     plt.xlabel('Column')
98     plt.ylabel('Row')
99     plt.show()
100
101 class ValueIteration(MarkovDecisionProcess):
102
103     def __init__(self, m=None, goal=None):
104         super().__init__(m, goal)
105         self.transition_value = {node: 10 if node == self.goal else 0 for node in self.m.grid}
106         self.transition_reward = {node: 100 if node == self.goal else 0 for node in self.m.grid}
107
108     def execute_iteration(self):
109         start = time.time()
110         self.start_memory_tracing()
111         values_converged_flag = False
112
113         while not values_converged_flag:

```

```

114     values_converged_flag = True
115
116     for current_node in self.m.grid:
117         temp_transition_value = []
118
119         for direction in ['N', 'S', 'W', 'E']:
120             if self.m.maze_map[current_node][direction] == 1:
121                 try:
122                     if direction == 'N':
123                         next_node = (current_node[0] - 1, current_node[1])
124                     elif direction == 'S':
125                         next_node = (current_node[0] + 1, current_node[1])
126                     elif direction == 'W':
127                         next_node = (current_node[0], current_node[1] - 1)
128                     elif direction == 'E':
129                         next_node = (current_node[0], current_node[1] + 1)
130                 except IndexError:
131                     next_node = None
132             if next_node is not None:
133                 next_transition_value = self.transition_probability [
134                     direction ] * (
135                         self.transition_reward [current_node] +
136                         self.discount_factor * self.transition_value [
137                             next_node])
138             temp_transition_value.append( next_transition_value )
139             self.transition_dictionary [current_node][ direction ] =
140                 next_transition_value
141             best_transition_value = max( temp_transition_value )
142
143             if abs( best_transition_value - self.transition_value [current_node] ) >
144                 self.convergence_threshold:
145                 values_converged_flag = False
146                 self.transition_value [current_node] = best_transition_value
147
148     end = time.time()
149     self.stop_memory_tracing()
150     self.cost_time = end - start
151
152     def get_direction_for_current_node ( self , current_node):
153         return get_direction ( self . transition_dictionary [current_node])

```

## E Code of Policy Iteration

```

1 from queue import Queue, PriorityQueue
2 import time
3 import tracemalloc as memory_trace
4 from math import sqrt
5 import numpy as np
6 import matplotlib.pyplot as plt
7 def start_memory_tracing():
8     memory_trace.stop()
9     memory_trace.start()
10
11 def get_direction (current_node):
12     return max(current_node, key=current_node.get)
13
14 class MarkovDecisionProcess:
15     def __init__ ( self , m=None, goal=None):
16         self.m = m
17         self.cost_time = 0
18         self.memory_peak = 0
19         self.final_path = {}
20         self.m = m
21         if goal is None:
22             raise AssertionError("Goal Cannot Be None")
23         self.goal = goal
24         self.start_node = ( self.m.rows, self.m.cols)
25
26         self.discount_factor = 0.9
27         self.convergence_threshold = 0.000001

```

```

28     self . transition_probability = { 'N': 1, 'S': 1, 'W': 1, 'E': 1}
29     self . transition_dictionary = {key: {subkey: 0 for subkey in self .m.maze_map[
30         key]} for key in self .m.maze_map}
31
32     def start_memory_tracing( self ):
33         memory_trace.stop()
34         memory_trace.start ()
35
36     def stop_memory_tracing(self):
37         _, self .memory_peak = memory_trace.get_traced_memory()
38         return self .memory_peak
39
40     def get_final_path ( self ):
41         next_node = [ self . start_node ]
42
43         while len(next_node) > 0:
44             current_node = next_node.pop()
45             if current_node == self .goal:
46                 break
47
48             direction = self . get_direction_for_current_node (current_node)
49
50             if direction == 'N':
51                 _temp_next_node_ = (current_node[0] - 1, current_node [1])
52             elif direction == 'S':
53                 _temp_next_node_ = (current_node[0] + 1, current_node [1])
54             elif direction == 'W':
55                 _temp_next_node_ = (current_node [0], current_node [1] - 1)
56             elif direction == 'E':
57                 _temp_next_node_ = (current_node [0], current_node [1] + 1)
58             else :
59                 raise AssertionError(" Invalid Direction ")
60             self . final_path [current_node] = _temp_next_node_
61             next_node.append(_temp_next_node_)
62
63         return self . final_path , self .cost_time , self .memory_peak
64
65     def get_direction_for_current_node ( self , current_node):
66         pass
67
68     def execute_iteration ( self ):
69         pass
70
71     def plot_maze_weights( self ):
72         weights = np.zeros(( self .m.rows, self .m.cols))
73         data = self . transition_value
74         for node, value in data.items():
75             weights[node[0] - 1, node[1] - 1] = value
76
77         plt . figure ( figsize =(10, 8))
78
79         if self .m.rows <= 30 and self .m.cols <= 30:
80             for i in range( self .m.rows):
81                 for j in range( self .m.cols):
82                     if self .__class__. __name__ == 'ValueIteration ':
83                         if self .m.rows > 20 or self .m.cols > 20:
84                             plt . text (j, i, str (int (weights[i, j])), ha='center' , va='center' , color='black')
85                         else :
86                             plt . text (j, i, format (weights[i, j], '.2f') , ha='center' , va='center' , color='black')
87                     else :
88                         if self .m.rows > 20 or self .m.cols > 20:
89                             plt . text (j, i, format (weights[i, j], '.1f') , ha='center' , va='center' , color='black')
90                         else :
91                             plt . text (j, i, format (weights[i, j], '.4f') , ha='center' , va='center' , color='black')
92
93         plt .imshow(weights, cmap='cool' , interpolation ='nearest' , origin='upper')
94         plt .colorbar ( label='Weight')
95         plt .xticks (np.arange( self .m.cols), np.arange(1, self .m.cols + 1))

```

```

95     plt . yticks (np.arange( self .m.rows), np.arange(1, self .m.rows + 1))
96     plt . title (f'Maze_Weights_Visualization_for_{ self .__class__.name}_'
97                 Algorithm_in_{self.m.rows}x{self.m.cols}Maze')
98     plt . xlabel ('Column')
99     plt . ylabel ('Row')
100    plt .show()

101   class PolicyIteration (MarkovDecisionProcess):

102       def __init__ ( self , m=None, goal=None):
103           super().__init__(m, goal)
104           self . transition_value = {node: 1 if node == self .goal else 0 for node in
105                                     self .m.grid}
106           self . transition_reward = {node: 1 if node == self .goal else 0 for node in
107                                      self .m.grid}
108           self . policy = {node: None if node == self .goal else 'N' for node in self .m
109                           .grid}

110       def calculate_transition_value ( self , current_node):
111           _temp_transition_value_ = {'N': 0, 'S': 0, 'W': 0, 'E': 0}
112           temp_node_transition_value = {current_node: _temp_transition_value_ }
113           for direction in ['N', 'S', 'W', 'E']:
114               if self .m.maze_map[current_node][direction] == 1:
115                   next_node = None
116                   try :
117                       if direction == 'N':
118                           next_node = (current_node[0] - 1, current_node [1])
119                       elif direction == 'S':
120                           next_node = (current_node[0] + 1, current_node [1])
121                       elif direction == 'W':
122                           next_node = (current_node [0], current_node [1] - 1)
123                       elif direction == 'E':
124                           next_node = (current_node [0], current_node [1] + 1)
125                   except IndexError:
126                       next_node = None
127                   if next_node is not None:
128                       next_transition_value = self . transition_value [next_node]
129                   else :
130                       next_transition_value = 0
131
132                   temp_node_transition_value [current_node ][ direction ] = self .
133                     transition_probability [ direction ] * (
134                         self . transition_reward [current_node] +
135                         self . discount_factor * next_transition_value )
136
137       return temp_node_transition_value

138       def execute_iteration ( self ):
139           start = time.time()
140           self .start_memory_tracing()
141           value_converged_flag = False
142           policy_converged_flag = False
143
144           while not policy_converged_flag :
145               policy_converged_flag = True
146               value_converged_flag = False
147               while not value_converged_flag :
148                   value_converged_flag = True
149                   for current_node in self .m.grid:
150                       if current_node == self .goal:
151                           continue
152                       current_policy = self .policy [current_node]
153                       if self .m.maze_map[current_node][current_policy] == 1:
154                           next_node = None
155                           try :
156                               if current_policy == 'N':
157                                   next_node = (current_node[0] - 1, current_node [1])
158                               elif current_policy == 'S':
159                                   next_node = (current_node[0] + 1, current_node [1])
160                               elif current_policy == 'W':
161                                   next_node = (current_node [0], current_node [1] - 1)
162                               elif current_policy == 'E':
163                                   next_node = (current_node [0], current_node [1] + 1)

```

```

162     except IndexError:
163         next_node = None
164     if next_node is not None:
165         next_transition_value = self.transition_value[next_node]
166     else:
167         next_transition_value = 0
168
169     self.transition_dictionary[current_node][current_policy] =
170     (
171         self.transition_probability[current_policy] * (
172             self.transition_reward[current_node] +
173             next_transition_value * self.discount_factor))
174
175     if abs(self.transition_value[current_node] - (self.
176         transition_probability[current_policy] * (
177             self.transition_reward[
178                 current_node] + next_transition_value * self.
179                 discount_factor))) > self.
180                 convergence_threshold:
181         self.transition_value[current_node] = self.
182             transition_probability[current_policy] * (
183                 self.transition_reward[
184                     current_node] + next_transition_value *
185                     self.discount_factor)
186
187     value_converged_flag = False
188
189     for current_node in self.m.grid:
190         if current_node == self.goal:
191             continue
192         current_node_transition_value = self.calculate_transition_value(
193             current_node)
194         current_node_transition_policy = get_direction(
195             current_node_transition_value[current_node])
196         if current_node_transition_policy != self.policy[current_node]:
197             self.policy[current_node] = current_node_transition_policy
198             policy_converged_flag = False
199
200         self.cost_time = time.time() - start
201         self.stop_memory_tracing()
202
203     def get_direction_for_current_node(self, current_node):
204         return self.policy[current_node]

```

## F Origin Data

For 5x5 Maze:Path Lengths: {'BFS': 9, 'DFS': 11, 'A\*': 9, 'VAL': 9, 'POL': 9}  
 Calculated Times: {'BFS': 0.2191, 'DFS': 0.0348, 'A\*': 0.1507, 'VAL': 12.697, 'POL': 3.583}  
 Memory Consumptions: {'BFS': 0.004, 'DFS': 0.001, 'A\*': 0.005, 'VAL': 0.002, 'POL': 0.002}  
 Visited Counts: {'BFS': 26, 'DFS': 22, 'A\*': 15}

For 20x20 Maze:Path Lengths: {'BFS': 39, 'DFS': 45, 'A\*': 39, 'VAL': 39, 'POL': 39}  
 Calculated Times: {'BFS': 2.3408, 'DFS': 0.1132, 'A\*': 0.4311, 'VAL': 214.5798, 'POL': 102.3591}  
 Memory Consumptions: {'BFS': 0.061, 'DFS': 0.005, 'A\*': 0.02, 'VAL': 0.034, 'POL': 0.022}  
 Visited Counts: {'BFS': 401, 'DFS': 90, 'A\*': 49}

For 30x30 Maze:Path Lengths: {'BFS': 59, 'DFS': 63, 'A\*': 61, 'VAL': 59, 'POL': 59}  
 Calculated Times: {'BFS': 5.2621, 'DFS': 0.1237, 'A\*': 3.4239, 'VAL': 487.3381, 'POL': 334.6488}  
 Memory Consumptions: {'BFS': 0.09, 'DFS': 0.007, 'A\*': 0.101, 'VAL': 0.076, 'POL': 0.049}  
 Visited Counts: {'BFS': 901, 'DFS': 126, 'A\*': 326}

For 50x50 Maze:Path Lengths: {'BFS': 99, 'DFS': 123, 'A\*': 99, 'VAL': 99, 'POL': 99}  
 Calculated Times: {'BFS': 13.545, 'DFS': 0.2058, 'A\*': 2.1842, 'VAL': 1335.9559, 'POL': 1554.523}  
 Memory Consumptions: {'BFS': 0.244, 'DFS': 0.017, 'A\*': 0.104, 'VAL': 0.21, 'POL': 0.136}  
 Visited Counts: {'BFS': 2501, 'DFS': 255, 'A\*': 243}

For 50x100 Maze:Path Lengths: {'BFS': 149, 'DFS': 173, 'A\*': 151, 'VAL': 149, 'POL': 149}  
 Calculated Times: {'BFS': 24.23, 'DFS': 0.3977, 'A\*': 7.443, 'VAL': 2660.815, 'POL': 6087.6012}  
 Memory Consumptions: {'BFS': 0.962, 'DFS': 0.021, 'A\*': 0.323, 'VAL': 0.42, 'POL': 0.265}  
 Visited Counts: {'BFS': 5001, 'DFS': 346, 'A\*': 898}

## G License of pyamaze

Link to pyamaze: [pyamaze](#)

MIT License

Copyright (c) 2021 Muhammad Ahsan Naeem

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.

## H License of matplotlib

Link to matplotlib: [Matplotlib](#)

License agreement for matplotlib versions 1.3.0 and later

---

1. This LICENSE AGREEMENT is between the Matplotlib Development Team ("MDT"), and the Individual or Organization ("Licensee") accessing and otherwise using matplotlib software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, MDT hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib alone or in any derivative version, provided, however, that MDT's License Agreement and MDT's notice of copyright, i.e., "Copyright (c) 2012- Matplotlib Development Team; All Rights Reserved" are retained in matplotlib alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib .
4. MDT is making matplotlib available to Licensee on an "AS IS" basis. MDT MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, MDT MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. MDT SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB , OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between MDT and

Licensee. This License Agreement does not grant permission to use MDT trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib , Licensee agrees to be bound by the terms and conditions of this License Agreement.

License agreement for matplotlib versions prior to 1.3.0

=====

1. This LICENSE AGREEMENT is between John D. Hunter ("JDH"), and the Individual or Organization ("Licensee") accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, JDH hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib alone or in any derivative version, provided, however, that JDH's License Agreement and JDH's notice of copyright, i.e., "Copyright (c) 2002-2011 John D. Hunter; All Rights Reserved" are retained in matplotlib alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib.

4. JDH is making matplotlib available to Licensee on an "AS IS" basis. JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB , OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between JDH and Licensee. This License Agreement does not grant permission to use JDH trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## I License of numpy

Link to numpy: pyamaze

Copyright (c) 2005-2024, NumPy Developers.  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the NumPy Developers nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.