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

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 generater. The module pyamaze is created for the easy generation of random maze and apply different search algorithm efficiently[1]. 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[2]. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python[3]. To visulize the performance of different algorithms, these two open source libraries are used in this assignment when visulize 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 visulize 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.

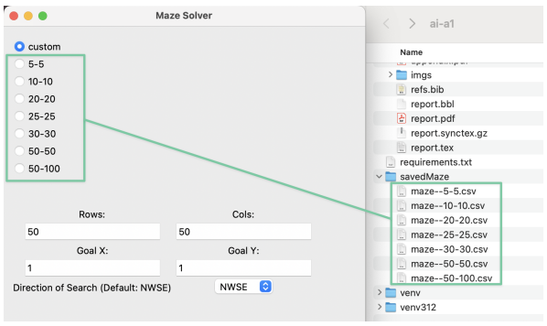


Figure 1: Auto Load Saved Maze From `savedMaze`

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.

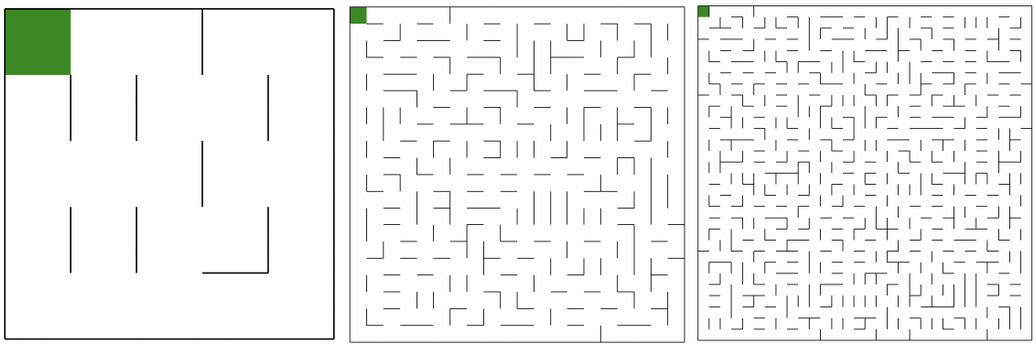
2.2 GUI

I've programed 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 calculated the path and the pyamaze will show the final path, and the performance of the algorithms will be shown in the GUI as label.

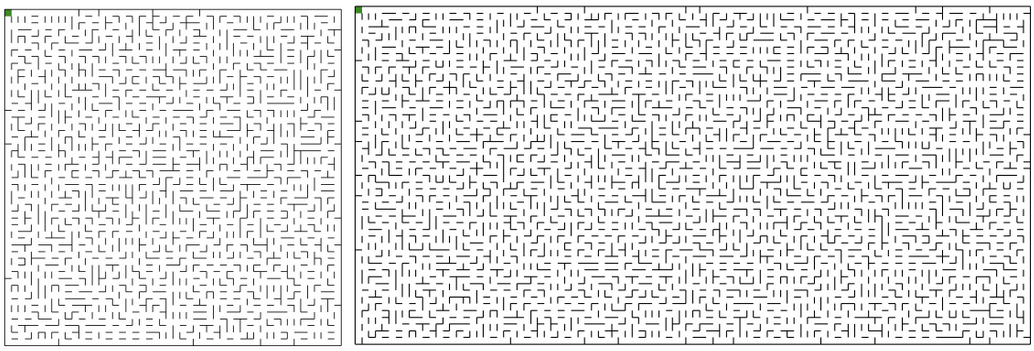
Besides, when select the algorithms, there will also be some figures shown from the matplotlib library to visulize the performance of the algorithms. The figure will show the path steps (includ the start node and the goal node), time cost(in milliseconds), and the memory peak(in MB) for each selected 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).

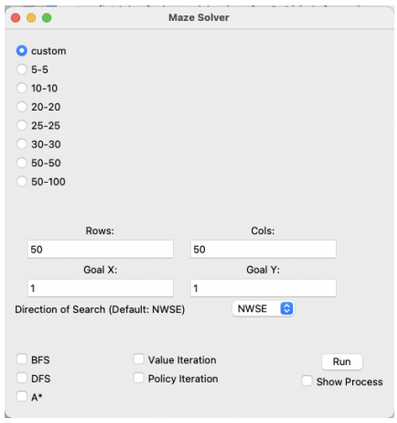


(a) 5x5 (b) 20x20 (c) 30x30



(d) 50x50 (e) 50x100

Figure 2: Used Maze



(a) main GUI

(b) labels in maze

Figure 3: GUI of the program

3.1 Graph Search Algorithms

`GraphSearchAlgorithms` is the 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[4] [5]. 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:

```
next_transition_value = self.transition_probability[direction] * (self.
    transition_reward[current_node] + self.discount_factor * self.transition_value[
    next_node])
temp_transition_value.append(next_transition_value)
self.transition_dictionary[current_node][direction] = next_transition_value
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:

```
temp_node_transition_value[current_node][direction] = self.transition_probability[direction] * (self.transition_reward[current_node] + self.discount_factor * next_transition_value)
self.transition_dictionary[current_node][current_policy] = (self.transition_probability[current_policy] * (self.transition_reward[current_node] + next_transition_value * self.discount_factor))
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:
    self.transition_value[current_node] = self.transition_probability[current_policy] * (self.transition_reward[current_node] + next_transition_value * self.discount_factor)
    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:

```
for current_node in self.m.grid:
if current_node == self.goal:
    continue
current_node_transition_value = self.calculate_transition_value(current_node)
current_node_transition_policy = get_direction(current_node_transition_value[current_node])
if current_node_transition_policy != self.policy[current_node]:
    self.policy[current_node] = current_node_transition_policy
    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, somtimes 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	**11**	**0.0348**	**0.001**	22
	A*	9	0.1507	0.005	**15**
	VAL	9	12.697	0.002	-
	POL	9	3.583	0.002	-
20x20	BFS	39	2.3408	0.061	401
	DFS	**45**	**0.1132**	**0.005**	90
	A*	39	0.4311	0.02	**49**
	VAL	39	214.5798	0.034	-
	POL	39	102.3591	0.022	-
30x30	BFS	59	5.2621	0.09	901
	DFS	**63**	**0.1237**	**0.007**	**126**
	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	**123**	**0.2058**	**0.017**	255
	A*	99	2.1842	0.104	**243**
	VAL	99	1335.9559	0.21	-
	POL	99	1554.523	0.136	-
50x100	BFS	149	24.23	0.962	5001
	DFS	**173**	**0.3977**	**0.021**	**346**
	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

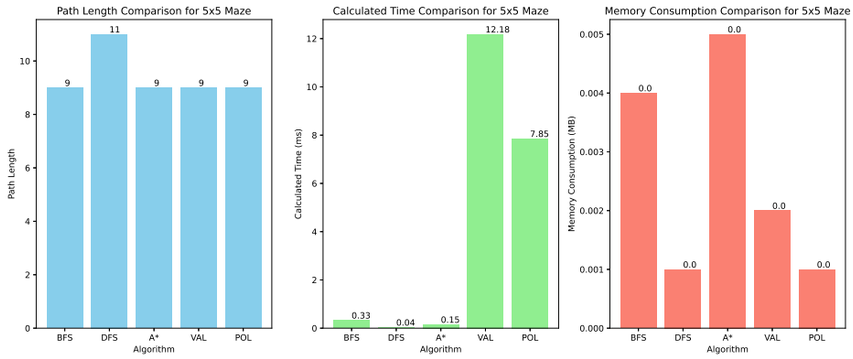
4.1 Path Steps, Time, and Memory Comparsion

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.

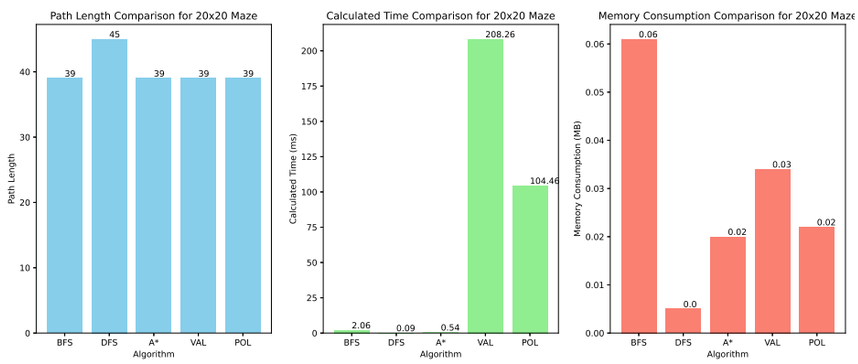
Comparing in MDP, the Policy Iteration is more efficient than the Value Iteration in terms of memory. When considering the time, the Value Iteration is more efficient than the Policy Iteration when the maze is smaller than 50x50. When the maze is bigger than 50x50, the Value Iteration is more efficient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

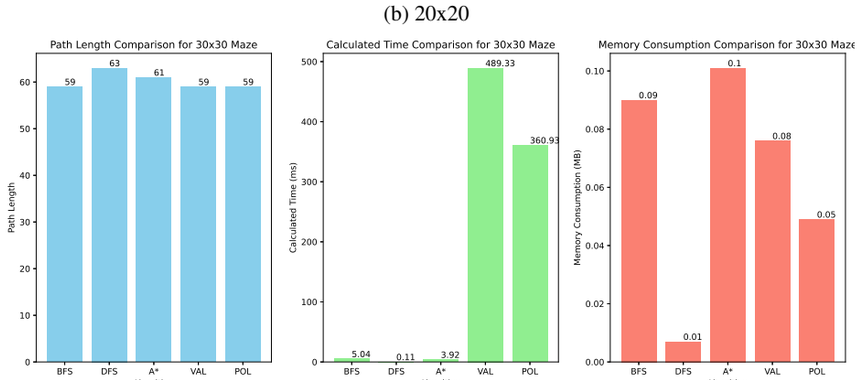
For mazes with multiple solutions, BFS, A*, Value Iteration and Policy Iteration have the shortest paths. DFS has the longest path. This is the expected result.



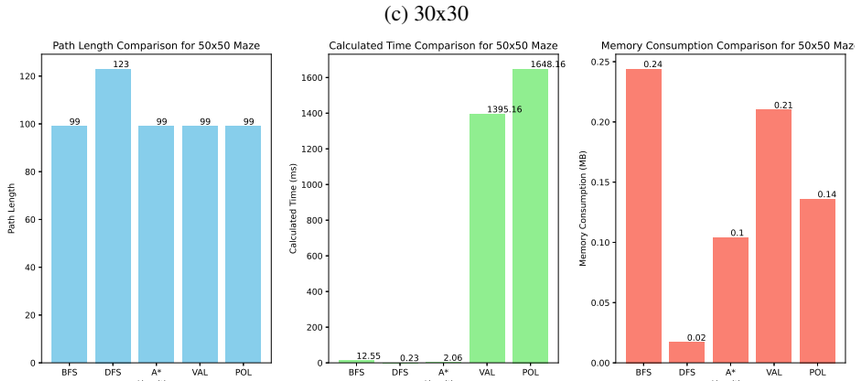
(a) 5x5



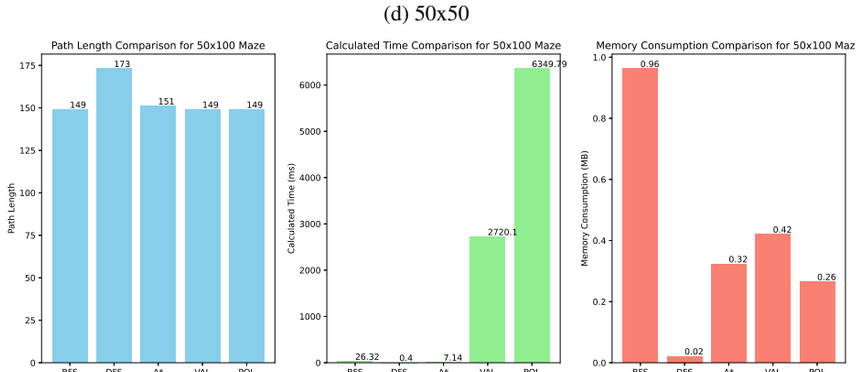
(b) 20x20



(c) 30x30



(d) 50x50

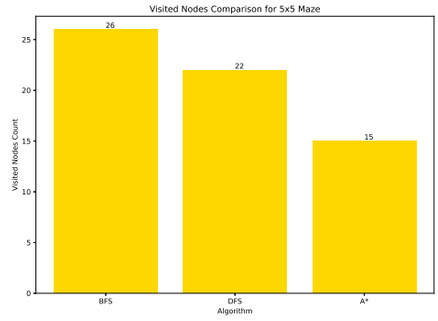


(e) 50x100

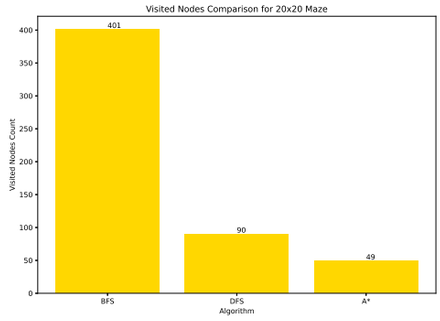
Figure 4: Path Steps, Time, and Memory Comparsion

4.2 Search Path Steps Comparsion for Graph Search Algorithms

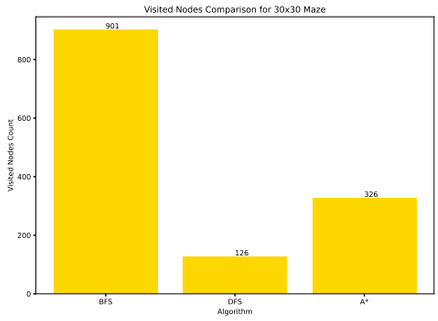
Figure 5 shows the search path steps comparsion of the graph search algorithms on the selected mazes.



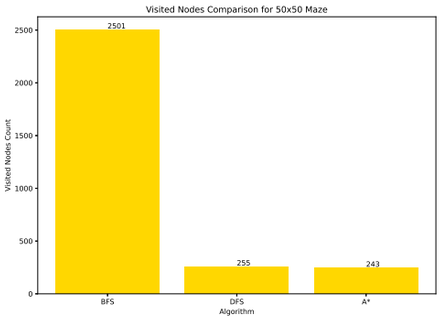
(a) 5x5



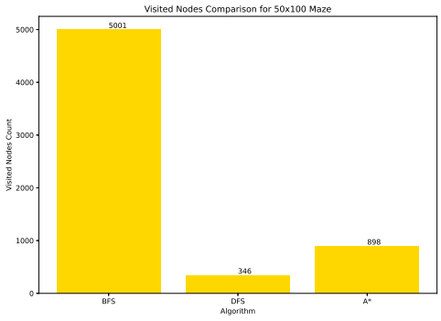
(b) 20x20



(c) 30x30



(d) 50x50



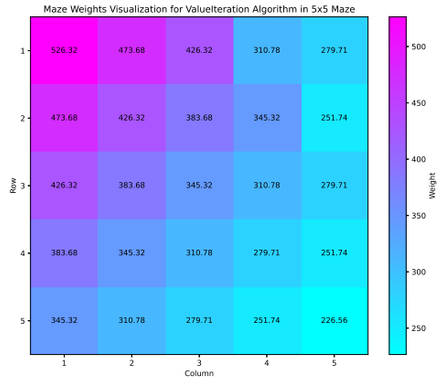
(e) 50x100

Figure 5: Search Steps Comparsion

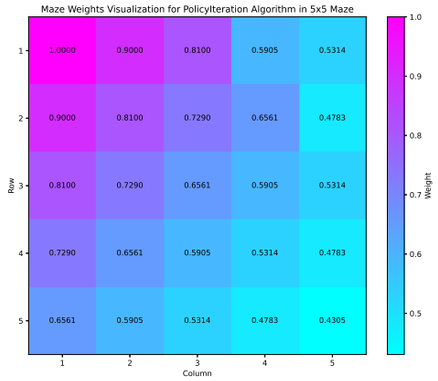
In most cases, A* and DFS explore the fewest nodes, BFS explores the most nodes.

4.3 Heat Map for MDP Algorithms

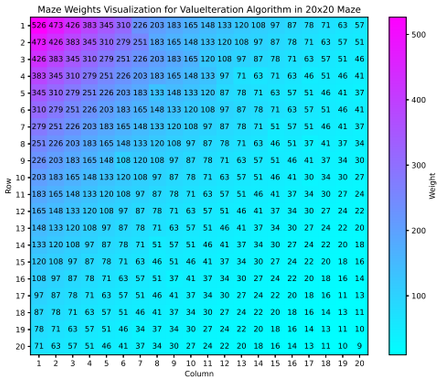
Figure 6 shows the heat map of the maze from MDP algorithms. This section is for visualizations.



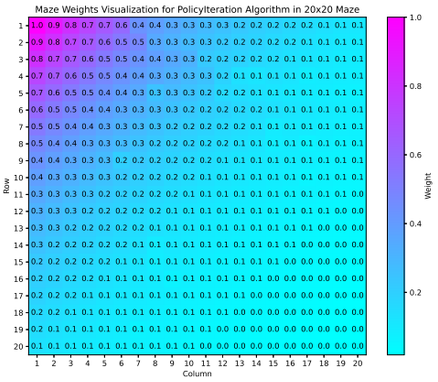
(a) 5x5 Value Iteration



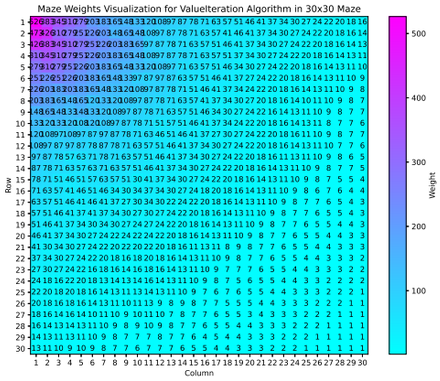
(b) 5x5 Policy Iteration



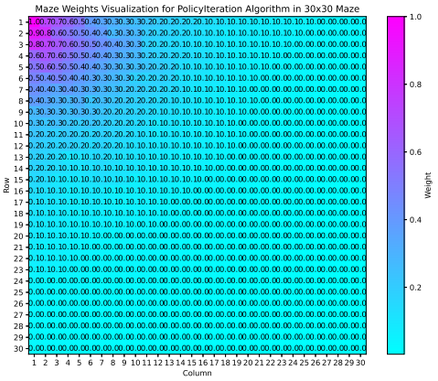
(c) 20x20 Value Iteration



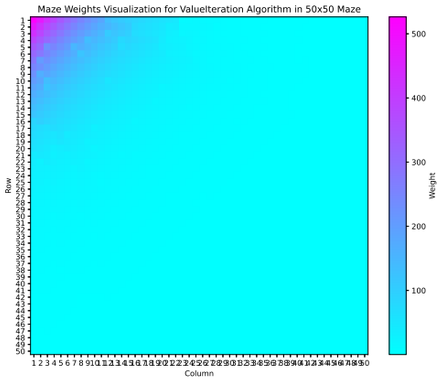
(d) 20x20 Policy Iteration



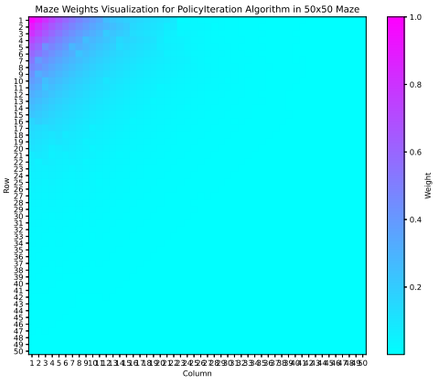
(e) 30x30 Value Iteration



(f) 30x30 Policy Iteration

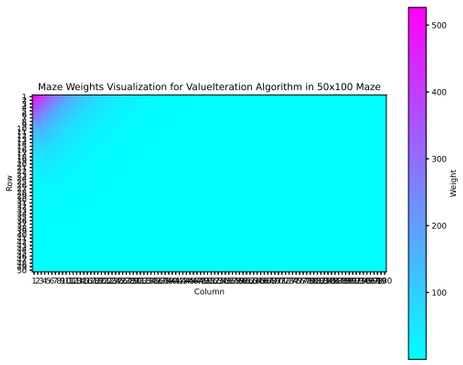


(g) 50x50 Value Iteration

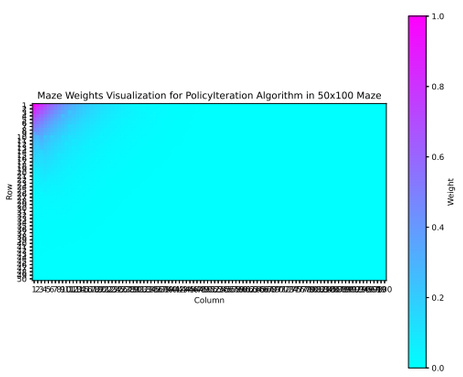


(h) 50x50 Policy Iteration

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

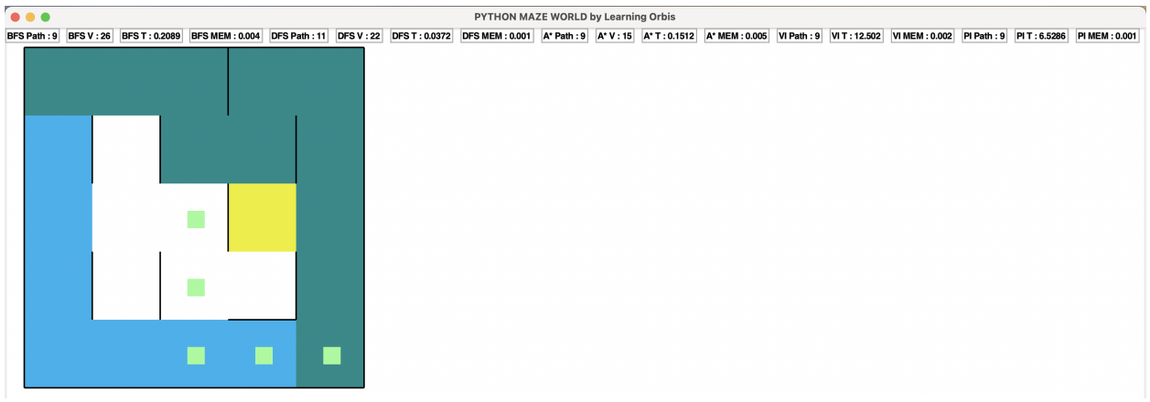
It can be concluded that it's important to consider which is the best algorithm in different situations. For example, if the maze is small and the goal is to find the shortest path, A* and BFS is the best choice. In other words, before using an algorithm, it is necessary to analyze the requirements of 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.

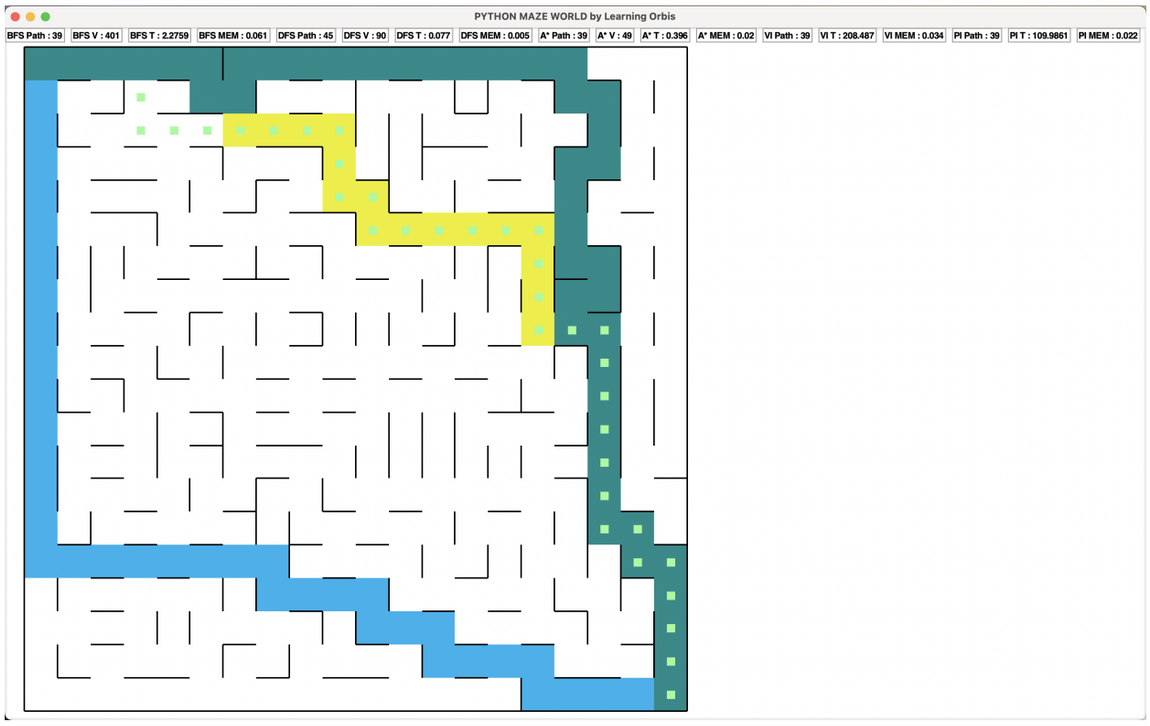
6 Soluctions of the Maze by All the Algorithms

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

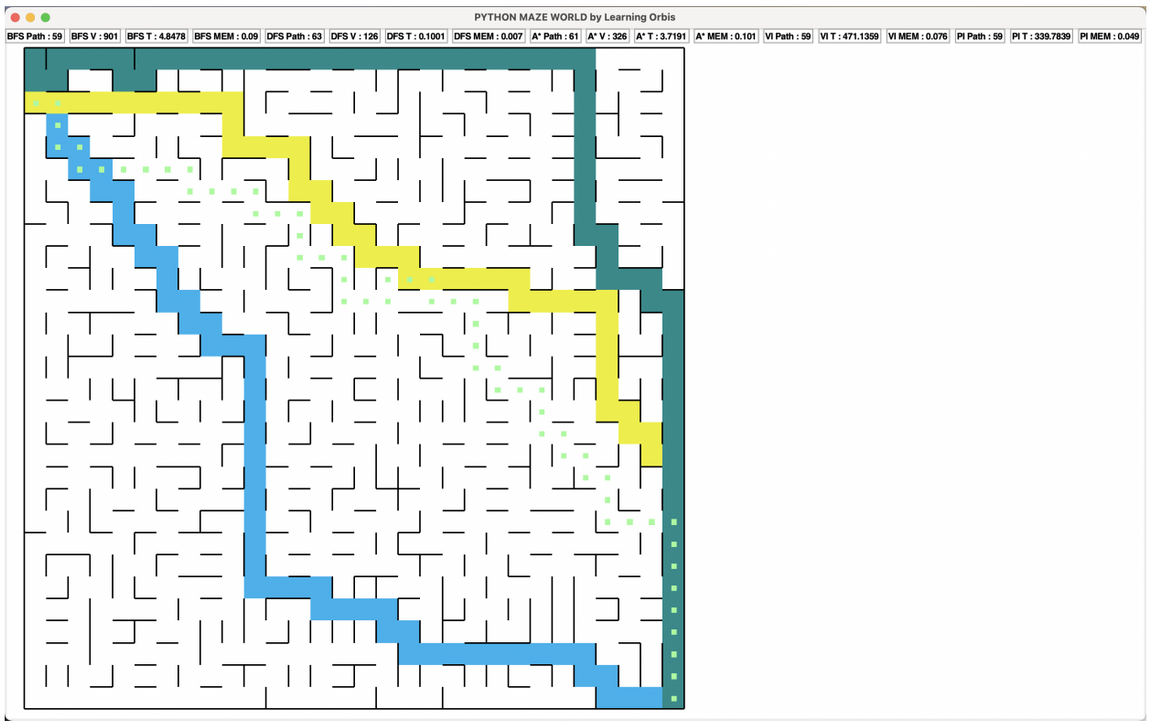
- The path with arrow is the explored/searched path, and the path with line or dots is the final path.
- `Yellow` color represents the final path of the algorithm `BFS`.
- `Cyan` color (dark green) represents the final path of the algorithm `DFS`.
- `Green` color (light green) represents the final path 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

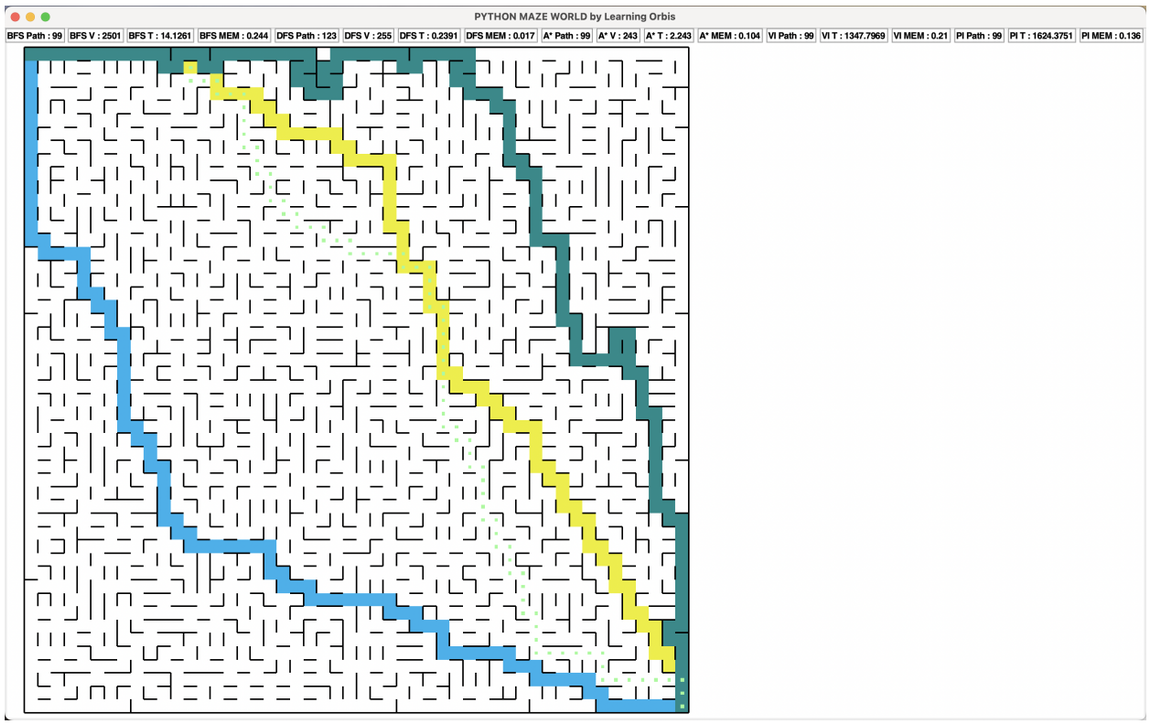


(b) 20x20

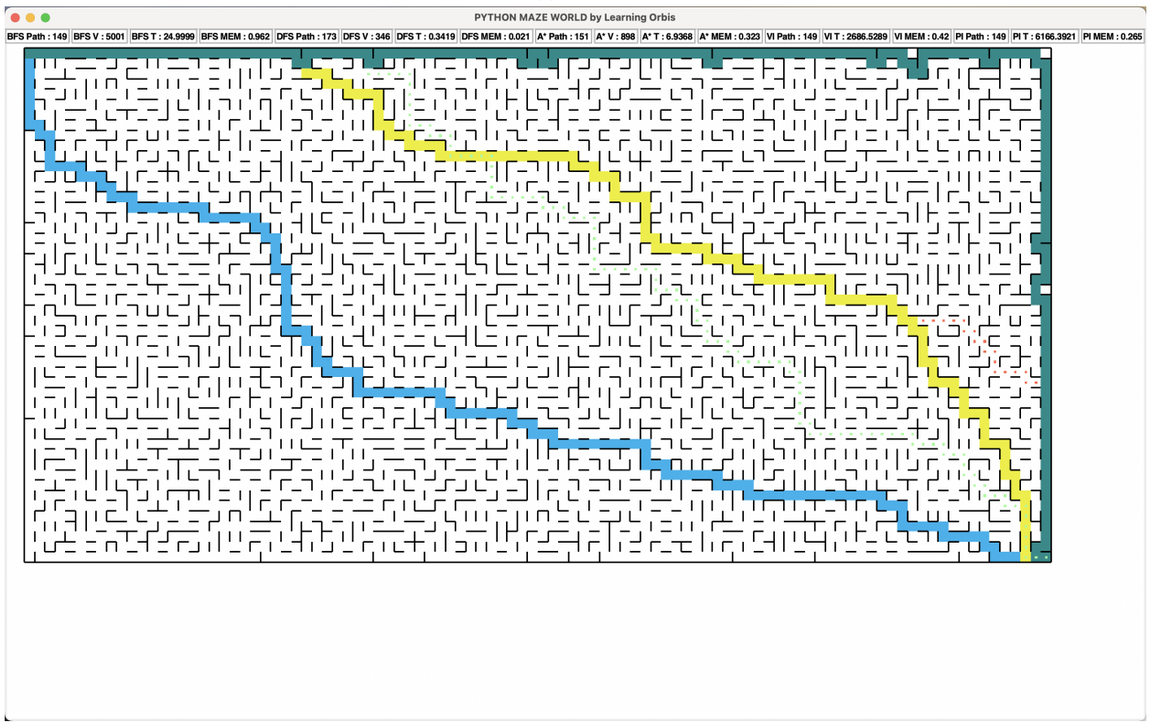


(c) 30x30

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



(a) 50x50



(b) 50x100

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

References

[1] MAN1986, "pyamaze: A python module to create and work on customizable random mazes - github." https://pypi.org/project/pyamaze/, 2022. Accessed: 2024-03-01.

[2] "Numpy: the fundamental package for scientific computing in python." https://numpy.org/doc/stable/user/whatisnumpy.html. Accessed: 2024-03-01.

[3] M. D. Team, "Matplotlib: Visualization with python." https://matplotlib.org/, 2024. Accessed: 2024-03-01.

[4] GeeksforGeeks, "A* search algorithm." https://www.geeksforgeeks.org/a-search-algorithm/, 2021.

[5] ratchet freak, "A* algorithm heuristic for maze solver - stack exchange." https://stackoverflow.com/questions/4938377/a-algorithm-heuristic-for-maze-solver, 2017.

A Code of BFS

```python
from queue import Queue, PriorityQueue
import time
import tracemalloc as memory_trace
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt

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

class GraphSearchAlgorithms:
    def __init__(self, m, goal):
        self.m = m
        self.cost_time = 0
        self.memory_peak = 0
        self.goal = goal
        self.start_node = (self.m.rows, self.m.cols)
        self.visited = set()
        self.nodes = 'NWSE'
        self.final_path = []
        self.search_path = []
        self.origin_dict = {}

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

    def validate_node(self, current_node):
        if current_node[0] < 1 or current_node[0] > self.m.rows or current_node[1] < 1
            or current_node[1] > self.m.cols:
            return False
        if current_node in self.visited:
            return False
        return True

    def compute_next_node(self, current_node):
        for direction in self.nodes:
            if self.m.maze_map[current_node][direction]:
                next_node = None
                if direction == 'N':
                    next_node = (current_node[0] - 1, current_node[1])
                elif direction == 'W':
                    next_node = (current_node[0], current_node[1] - 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)
                if self.validate_node(next_node):
                    return next_node

    def traceback_final_path(self):
        final_path = []
        current_node = self.goal
        while current_node != self.start_node:
            final_path.append(current_node)
            current_node = self.origin_dict[current_node]
        final_path.append(self.start_node)
        return final_path[::-1]

    def execute_algorithm(self):
        pass

    def get_final_path(self):
        return self.final_path, len(self.search_path), self.cost_time, self.
            memory_peak

class BFS(GraphSearchAlgorithms):
    def __init__(self, m, goal):
        super().__init__(m, goal)
```

```
        self.queue = Queue()
        self.queue.put(self.start_node)

    def execute_algorithm(self):
        start = time.time()
        start_memory_tracing()
        while self.queue:
            current_node = self.queue.get()
            self.search_path.append(current_node)
            if current_node == self.goal:
                self.final_path = self.traceback_final_path()
                end = time.time()
                self.stop_memory_tracing()
                self.cost_time = end - start
                return self.search_path, self.final_path
            while True:
                new_position = self.compute_next_node(current_node)
                if new_position:
                    self.queue.put(new_position)
                    self.visited.add(new_position)
                    self.origin_dict[new_position] = current_node
                else:
                    break
        return self.search_path, None
```

B Code of DFS

```
from queue import Queue, PriorityQueue
import time
import tracemalloc as memory_trace
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt

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

class GraphSearchAlgorithms:
    def __init__(self, m, goal):
        self.m = m
        self.cost_time = 0
        self.memory_peak = 0
        self.goal = goal
        self.start_node = (self.m.rows, self.m.cols)
        self.visited = set()
        self.nodes = 'NWSE'
        self.final_path = []
        self.search_path = []
        self.origin_dict = {}

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

    def validate_node(self, current_node):
        if current_node[0] < 1 or current_node[0] > self.m.rows or current_node[1] < 1 or
            current_node[1] > self.m.cols:
            return False
        if current_node in self.visited:
            return False
        return True

    def compute_next_node(self, current_node):
        for direction in self.nodes:
            if self.m.maze_map[current_node][direction]:
                next_node = None
                if direction == 'N':
                    next_node = (current_node[0] - 1, current_node[1])
                elif direction == 'W':
```

```
                    next_node = (current_node[0], current_node[1] - 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)
                if self.validate_node(next_node):
                    return next_node

    def traceback_final_path(self):
        final_path = []
        current_node = self.goal
        while current_node != self.start_node:
            final_path.append(current_node)
            current_node = self.origin_dict[current_node]
        final_path.append(self.start_node)
        return final_path[::-1]

    def execute_algorithm(self):
        pass

    def get_final_path(self):
        return self.final_path, len(self.search_path), self.cost_time, self.
            memory_peak

        class DFS(GraphSearchAlgorithms):
        def __init__(self, m, goal):
            super().__init__(m, goal)
            self.final_path = [self.start_node]
            self.search_path = [self.start_node]

        def execute_algorithm(self):
            start = time.time()
            start_memory_tracing()
            while self.final_path:
                current_position = self.final_path.pop()
                self.search_path.append(current_position)
                if current_position == self.goal:
                    end = time.time()
                    self.stop_memory_tracing()
                    self.cost_time = end - start
                    self.final_path.append(current_position)
                    return self.search_path, self.final_path
                self.visited.add(current_position)
                new_position = self.compute_next_node(current_position)
                if new_position:
                    self.final_path.append(current_position)
                    self.final_path.append(new_position)
                    self.search_path.append(new_position)
            return self.search_path, None
```

C Code of A*

```
from queue import Queue, PriorityQueue
import time
import tracemalloc as memory_trace
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt

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

class GraphSearchAlgorithms:
    def __init__(self, m, goal):
        self.m = m
        self.cost_time = 0
        self.memory_peak = 0
        self.goal = goal
        self.start_node = (self.m.rows, self.m.cols)
```

```
        self.visited = set()
        self.nodes = 'NWSE'
        self.final_path = []
        self.search_path = []
        self.origin_dict = {}

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

    def validate_node(self, current_node):
        if current_node[0] < 1 or current_node[0] > self.m.rows or current_node[1] < 1
            or current_node[1] > self.m.cols:
            return False
        if current_node in self.visited:
            return False
        return True

    def compute_next_node(self, current_node):
        for direction in self.nodes:
            if self.m.maze_map[current_node][direction]:
                next_node = None
                if direction == 'N':
                    next_node = (current_node[0] - 1, current_node[1])
                elif direction == 'W':
                    next_node = (current_node[0], current_node[1] - 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)
                if self.validate_node(next_node):
                    return next_node

    def traceback_final_path(self):
        final_path = []
        current_node = self.goal
        while current_node != self.start_node:
            final_path.append(current_node)
            current_node = self.origin_dict[current_node]
        final_path.append(self.start_node)
        return final_path[::-1]

    def execute_algorithm(self):
        pass

    def get_final_path(self):
        return self.final_path, len(self.search_path), self.cost_time, self.
            memory_peak

        class AStar(GraphSearchAlgorithms):
        def __init__(self, m, goal):
            super().__init__(m, goal)
            self.g_score = {self.start_node: 0}
            self.priority_queue = PriorityQueue()
            self.manhattan_flag = True

        def h_score(self, node, manhattan=True):
            if manhattan:
                return abs(node[0] - self.goal[0]) + abs(node[1] - self.goal[1])
            else:  # Euclidean
                return sqrt(((node[0] - self.goal[0]) ** 2) + ((node[1] - self.goal[1])
                    ** 2))

        def execute_algorithm(self):
            start = time.time()
            start_memory_tracing()
            self.priority_queue.put((self.g_score[self.start_node] + self.h_score(self.
                start_node), self.start_node))

            while self.priority_queue:
                _, current_position = self.priority_queue.get()
                self.search_path.append(current_position)
```

```
                if current_position == self.goal:
                    end = time.time()
                    self.stop_memory_tracing()
                    self.cost_time = end - start
                    self.final_path = self.traceback_final_path()
                    return self.search_path, self.final_path
                self.visited.add(current_position)
                while True:
                    new_position = self.compute_next_node(current_position)
                    if new_position:
                        self.g_score[new_position] = self.g_score[current_position] + 1
                        self.priority_queue.put((self.g_score[new_position] + self.
                            h_score(new_position), new_position))
                        self.origin_dict[new_position] = current_position
                        self.visited.add(new_position)
                    else:
                        break
            return self.search_path, None
```

D Code of Value Iteration

```
from queue import Queue, PriorityQueue
import time
import tracemalloc as memory_trace
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
def start_memory_tracing():
    memory_trace.stop()
    memory_trace.start()

def get_direction(current_node):
    return max(current_node, key=current_node.get)

    class MarkovDecisionProcess:
    def __init__(self, m=None, goal=None):
        self.m = m
        self.cost_time = 0
        self.memory_peak = 0
        self.final_path = {}
        self.m = m
        if goal is None:
            raise AssertionError("Goal Cannot Be None")
        self.goal = goal
        self.start_node = (self.m.rows, self.m.cols)

        self.discount_factor = 0.9
        self.convergence_threshold = 0.000001
        self.transition_probability = {'N': 1, 'S': 1, 'W': 1, 'E': 1}
        self.transition_dictionary = {key: {subkey: 0 for subkey in self.m.maze_map[
            key]} for key in self.m.maze_map}

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

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

    def get_final_path(self):
        next_node = [self.start_node]

        while len(next_node) > 0:
            current_node = next_node.pop()
            if current_node == self.goal:
                break

            direction = self.get_direction_for_current_node(current_node)

```

```
                if direction == 'N':
                    _temp_next_node_ = (current_node[0] - 1, current_node[1])
                elif direction == 'S':
                    _temp_next_node_ = (current_node[0] + 1, current_node[1])
                elif direction == 'W':
                    _temp_next_node_ = (current_node[0], current_node[1] - 1)
                elif direction == 'E':
                    _temp_next_node_ = (current_node[0], current_node[1] + 1)
                else:
                    raise AssertionError("Invalid Direction")
                self.final_path[current_node] = _temp_next_node_
                next_node.append(_temp_next_node_)

        return self.final_path, self.cost_time, self.memory_peak

    def get_direction_for_current_node(self, current_node):
        pass

    def execute_iteration(self):
        pass

    def plot_maze_weights(self):
        weights = np.zeros((self.m.rows, self.m.cols))
        data = self.transition_value
        for node, value in data.items():
            weights[node[0] - 1, node[1] - 1] = value

        plt.figure(figsize=(10, 8))

        if self.m.rows <= 30 and self.m.cols <= 30:
            for i in range(self.m.rows):
                for j in range(self.m.cols):
                    if self.__class__.__name__ == 'ValueIteration':
                        if self.m.rows > 20 or self.m.cols > 20:
                            plt.text(j, i, str(int(weights[i, j])), ha='center', va='
                                center', color='black')
                        else:
                            plt.text(j, i, format(weights[i, j], '.2f'), ha='center',
                                va='center', color='black')
                    else:
                        if self.m.rows > 20 or self.m.cols > 20:
                            plt.text(j, i, format(weights[i, j], '.1f'), ha='center',
                                va='center', color='black')
                        else:
                            plt.text(j, i, format(weights[i, j], '.4f'), ha='center',
                                va='center', color='black')

        plt.imshow(weights, cmap='cool', interpolation='nearest', origin='upper')
        plt.colorbar(label='Weight')
        plt.xticks(np.arange(self.m.cols), np.arange(1, self.m.cols + 1))
        plt.yticks(np.arange(self.m.rows), np.arange(1, self.m.rows + 1))
        plt.title(f'Maze Weights Visualization for {self.__class__.__name__}
            Algorithm in {self.m.rows}x{self.m.cols} Maze')
        plt.xlabel('Column')
        plt.ylabel('Row')
        plt.show()

class ValueIteration(MarkovDecisionProcess):

    def __init__(self, m=None, goal=None):
        super().__init__(m, goal)
        self.transition_value = {node: 10 if node == self.goal else 0 for node in self.
            m.grid}
        self.transition_reward = {node: 100 if node == self.goal else 0 for node in
            self.m.grid}

    def execute_iteration(self):
        start = time.time()
        self.start_memory_tracing()
        values_converged_flag = False

        while not values_converged_flag:
```

```
                values_converged_flag = True

                for current_node in self.m.grid:
                    temp_transition_value = []

                    for direction in ['N', 'S', 'W', 'E']:
                        if self.m.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 == 'W':
                                    next_node = (current_node[0], current_node[1] - 1)
                                elif direction == 'E':
                                    next_node = (current_node[0], current_node[1] + 1)
                            except IndexError:
                                next_node = None
                            if next_node is not None:
                                next_transition_value = self.transition_probability[
                                    direction] * (
                                        self.transition_reward[current_node] +
                                        self.discount_factor * self.transition_value[
                                            next_node])
                                temp_transition_value.append(next_transition_value)
                                self.transition_dictionary[current_node][direction] =
                                    next_transition_value
                    best_transition_value = max(temp_transition_value)

                    if abs(best_transition_value - self.transition_value[current_node]) >
                        self.convergence_threshold:
                        values_converged_flag = False
                        self.transition_value[current_node] = best_transition_value

            end = time.time()
            self.stop_memory_tracing()
            self.cost_time = end - start

        def get_direction_for_current_node(self, current_node):
            return get_direction(self.transition_dictionary[current_node])
```

E Code of Policy Iteration

```
from queue import Queue, PriorityQueue
import time
import tracemalloc as memory_trace
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
def start_memory_tracing():
    memory_trace.stop()
    memory_trace.start()

def get_direction(current_node):
    return max(current_node, key=current_node.get)

    class MarkovDecisionProcess:
    def __init__(self, m=None, goal=None):
        self.m = m
        self.cost_time = 0
        self.memory_peak = 0
        self.final_path = {}
        self.m = m
        if goal is None:
            raise AssertionError("Goal Cannot Be None")
        self.goal = goal
        self.start_node = (self.m.rows, self.m.cols)

        self.discount_factor = 0.9
        self.convergence_threshold = 0.000001
```

```
        self.transition_probability = {'N': 1, 'S': 1, 'W': 1, 'E': 1}
        self.transition_dictionary = {key: {subkey: 0 for subkey in self.m.maze_map[
            key]} for key in self.m.maze_map}

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

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

    def get_final_path(self):
        next_node = [self.start_node]

        while len(next_node) > 0:
            current_node = next_node.pop()
            if current_node == self.goal:
                break

            direction = self.get_direction_for_current_node(current_node)

            if direction == 'N':
                _temp_next_node_ = (current_node[0] - 1, current_node[1])
            elif direction == 'S':
                _temp_next_node_ = (current_node[0] + 1, current_node[1])
            elif direction == 'W':
                _temp_next_node_ = (current_node[0], current_node[1] - 1)
            elif direction == 'E':
                _temp_next_node_ = (current_node[0], current_node[1] + 1)
            else:
                raise AssertionError("Invalid Direction")
            self.final_path[current_node] = _temp_next_node_
            next_node.append(_temp_next_node_)

        return self.final_path, self.cost_time, self.memory_peak

    def get_direction_for_current_node(self, current_node):
        pass

    def execute_iteration(self):
        pass

    def plot_maze_weights(self):
        weights = np.zeros((self.m.rows, self.m.cols))
        data = self.transition_value
        for node, value in data.items():
            weights[node[0] - 1, node[1] - 1] = value

        plt.figure(figsize=(10, 8))

        if self.m.rows <= 30 and self.m.cols <= 30:
            for i in range(self.m.rows):
                for j in range(self.m.cols):
                    if self.__class__.__name__ == 'ValueIteration':
                        if self.m.rows > 20 or self.m.cols > 20:
                            plt.text(j, i, str(int(weights[i, j])), ha='center', va='
                                center', color='black')
                        else:
                            plt.text(j, i, format(weights[i, j], '.2f'), ha='center',
                                va='center', color='black')
                    else:
                        if self.m.rows > 20 or self.m.cols > 20:
                            plt.text(j, i, format(weights[i, j], '.1f'), ha='center',
                                va='center', color='black')
                        else:
                            plt.text(j, i, format(weights[i, j], '.4f'), ha='center',
                                va='center', color='black')

        plt.imshow(weights, cmap='cool', interpolation='nearest', origin='upper')
        plt.colorbar(label='Weight')
        plt.xticks(np.arange(self.m.cols), np.arange(1, self.m.cols + 1))
```

```
        plt.yticks(np.arange(self.m.rows), np.arange(1, self.m.rows + 1))
        plt.title(f'Maze Weights Visualization for {self.__class__.__name__} Algorithm in {self.m.rows}x{self.m.cols} Maze')
        plt.xlabel('Column')
        plt.ylabel('Row')
        plt.show()

    class PolicyIteration(MarkovDecisionProcess):

    def __init__(self, m=None, goal=None):
        super().__init__(m, goal)
        self.transition_value = {node: 1 if node == self.goal else 0 for node in self.m.grid}
        self.transition_reward = {node: 1 if node == self.goal else 0 for node in self.m.grid}
        self.policy = {node: None if node == self.goal else 'N' for node in self.m.grid}

    def calculate_transition_value(self, current_node):
        _temp_transition_value_ = {'N': 0, 'S': 0, 'W': 0, 'E': 0}
        temp_node_transition_value = {current_node: _temp_transition_value_}
        for direction in ['N', 'S', 'W', 'E']:
            if self.m.maze_map[current_node][direction] == 1:
                next_node = None
                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 == 'W':
                        next_node = (current_node[0], current_node[1] - 1)
                    elif direction == 'E':
                        next_node = (current_node[0], current_node[1] + 1)
                except IndexError:
                    next_node = None
                if next_node is not None:
                    next_transition_value = self.transition_value[next_node]
                else:
                    next_transition_value = 0

                temp_node_transition_value[current_node][direction] = self.transition_probability[direction] * (
                    self.transition_reward[current_node] +
                    self.discount_factor * next_transition_value)
        return temp_node_transition_value

    def execute_iteration(self):
        start = time.time()
        self.start_memory_tracing()
        value_converged_flag = False
        policy_converged_flag = False

        while not policy_converged_flag:
            policy_converged_flag = True
            value_converged_flag = False
            while not value_converged_flag:
                value_converged_flag = True
                for current_node in self.m.grid:
                    if current_node == self.goal:
                        continue
                    current_policy = self.policy[current_node]
                    if self.m.maze_map[current_node][current_policy] == 1:
                        next_node = None
                        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 == 'W':
                                next_node = (current_node[0], current_node[1] - 1)
                            elif current_policy == 'E':
                                next_node = (current_node[0], current_node[1] + 1)
```

```
                            except IndexError:
                                next_node = None
                            if next_node is not None:
                                next_transition_value = self.transition_value[next_node]
                            else:
                                next_transition_value = 0

                            self.transition_dictionary[current_node][current_policy] = (
                                self.transition_probability[current_policy] * (
                                self.transition_reward[current_node] +
                                next_transition_value * self.discount_factor))

                            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:
                                self.transition_value[current_node] = self.
                                transition_probability[current_policy] * (
                                self.transition_reward[
                                current_node] + next_transition_value *
                                self.discount_factor)
                                value_converged_flag = False
                    for current_node in self.m.grid:
                        if current_node == self.goal:
                            continue
                        current_node_transition_value = self.calculate_transition_value(
                            current_node)
                        current_node_transition_policy = get_direction(
                            current_node_transition_value[current_node])
                        if current_node_transition_policy != self.policy[current_node]:
                            self.policy[current_node] = current_node_transition_policy
                            policy_converged_flag = False

                self.cost_time = time.time() - start
                self.stop_memory_tracing()

            def get_direction_for_current_node(self, current_node):
                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.
```