Assignment 1 – Search and MDPs Algorithm
Module: CS7IS2-202324 Artificial Intelligence

Student Name: Indrajeet Kumar
Student Number: 23345983

1. Introduction

In this assignment, we will explore different search and decision-making algorithms in the context of navigating mazes. First, we'll make a maze creator that can make mazes of different sizes. Then, we'll use various methods like Breadth First Search, Depth First Search, A* Star Algorithm, and some Markov Decision Processes (MDP) algorithms to solve these mazes. In the end, we'll summarise the findings, evaluate how well each algorithm performed, and make judgements regarding how useful each method is for navigating mazes.

1. Maze Implementation

For this assignment, Pyamaze [1], an open-source Python library known for its capacity to create mazes of various sizes, was used as the maze generation tool. Using 'Pyamaze' it was possible to comply with the task's main criterion, which was to create mazes of different sizes. Notably, Pyamaze gives users a great deal of freedom when creating mazes with unique features. These features include the ability to create mazes with patterns that are either horizontal or vertical, as well as the ability to create flawless mazes with one path to the objective or several pathways. Moreover, Pyamaze gives users the freedom to choose the maze's beginning point and finish, or target state, as they see fit.
Figure 1 displays a 5x5 maze created using Pyamaze, with the agent positioned at coordinates (5,5) and the target at (1,1) marked in green.

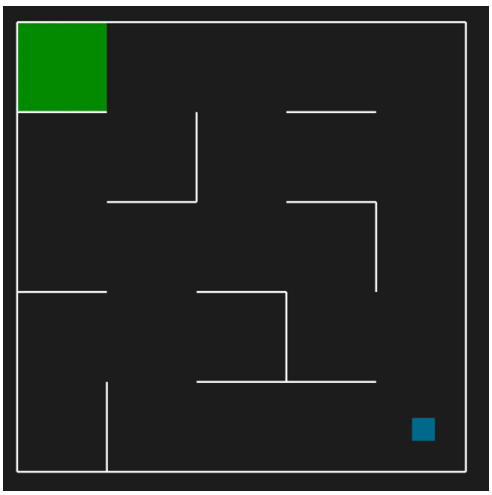

Figure 1 : 5x5 maze generated by pyamaze

2. Search Algorithms

II Depth First Search (DFS)

Depth First Search (DFS) is a search algorithm that prioritizes exploring the deepest unexplored nodes first. It operates using a Last In First Out (LIFO) strategy, similar to how a stack works. Beginning from the initial node, DFS thoroughly investigates each branch before backtracking. This process continues recursively until it reaches the goal state [2].

DFS is particularly effective when the solution lies deeper within the search space. However, it may not be the best choice if the target state is closer to the starting point. Additionally, DFS may encounter difficulties in situations involving infinite depth spaces or loops within the search area.

The code uses Depth First Search (DFS) to navigate a maze grid. It starts from the bottom-right corner and tries to reach the top-left corner. It explores paths one by one, backtracking if it hits a dead-end. Once it reaches the target, it shows the path it took using an arrow moving through the maze. Then, it runs to display the path in action as we can see in figure 2.

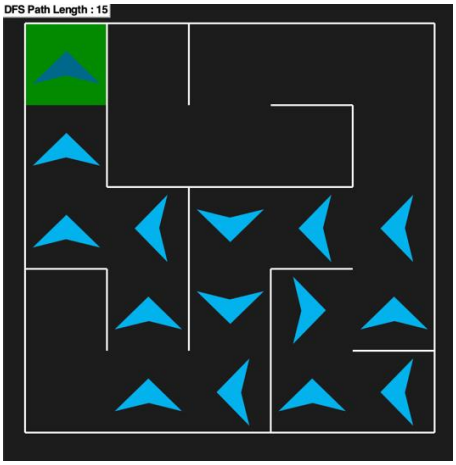

Figure 2 : DFS on 5x5 maze

II Breadth First Search (BFS)

Breadth First Search (BFS) is different from Depth First Search (DFS). It starts by exploring all nodes at one level before moving to the next. This method uses a "First In First Out" (FIFO) strategy. BFS looks at the closest nodes first, so new options wait their turn at the end of the line. It's good at finding solutions when we have a limited number of options and each step costs the same. But if the answer is deep down in the options, BFS might not be the best pick. Also, BFS needs a lot of memory, which can be a problem with big searches [2]. The figure below (Figure 3) shows the path of an agent from the bottom-right corner tries to reach the top-left corner in a 5x5 maze.

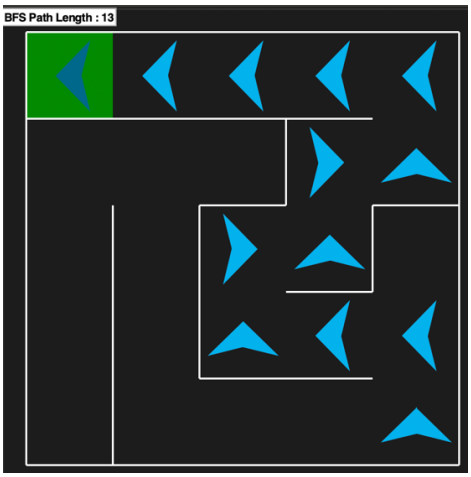

Figure 3 : BFS on 5x5 maze

II A Star Search Algorithm (A*)

A Star Search Algorithm (A*) is a smart way to find the best path in a maze or a network of connected points. It's like combining two other methods called Uniform Cost Search and Greedy Best First Search. A* uses a special formula that looks at how far we've already traveled (using Uniform Cost Search) and how close we are to our goal (using Greedy Best First Search). This formula helps A* to pick the best next step. The great thing about A* is that it's good at avoiding dead ends and it tries to avoid paths that are already looking expensive. A* works by calculating the cost of each step using something called a heuristic. In this assignment, we're using something called Manhattan distance as our heuristic. This is because it's a good guess of how far we are from our goal, and it's faster than another method called Euclidean distance.

The code demonstrates the implementation of the A* algorithm to find the optimal path in a maze environment using the pyamaze [2] library. The h function calculates the Manhattan distance heuristic between two cells in the maze. The aStar function initializes g_score and f_score

dictionaries to store the cost values for each cell and employs a priority queue to explore cells based on their heuristic values. It iteratively selects the cell with the lowest f_score and updates its neighbours' g_score and f_score if a shorter path is found. The algorithm maintains a dictionary aPath to keep track of the optimal path. In the main section, a 5x5 maze is created and generated, and the A* algorithm is applied to find the optimal path from the bottom-right corner to the top-left corner. The path is then traced and displayed using an agent in the maze, along with a text label indicating the length of the path, before running the maze.

The figure below (Figure 6) shows the path of an agent from the start node (5, 5) to the goal node (1, 1) in a 5x5 maze. Additionally, Pseudocode is also shown below for this A* maze search as shown in figure 4.

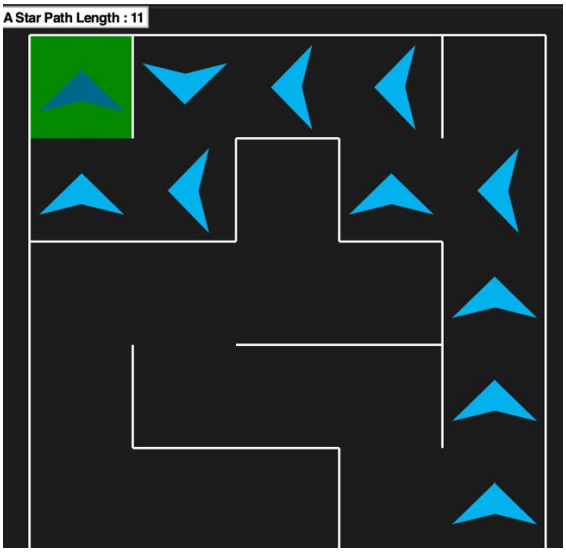

Figure 4 : A star (A) on 5x5 maze*

3. MARKOV DECISION PROCESS (MDP)

A Markov Decision Process (MDP) is a type of problem where we make decisions in a world that might change in unpredictable ways. In this world, you have different situations or "states," and for each situation, we can take different actions. These actions might lead us to different situations, and we also get some rewards for taking those actions.

The main goal of an MDP is to make decisions that will give us the most rewards on average. To do this, we need to figure out the best actions to take in each situation. We call this a "policy," which is just a set of instructions telling you what action to take in each situation.

There are two main ways to solve an MDP: Value Iteration and Policy Iteration. In Value Iteration, we keep track of how good each situation is and then use that information to figure out the best actions. In Policy Iteration, we directly try to find the best policy without worrying too much about the exact values of each situation. Both methods aim to find the best way to act in each situation to maximize your rewards in the long run.

I. VALUE ITERATIONS

Value Iteration works by figuring out how useful each state is, and then using that information to decide the best actions to take in each state. It does this by calculating the 'utility' of each state, which is like a measure of how good or valuable that state is.

To calculate the utility of a state, we use a formula called the Bellman Equation. This equation takes into account the reward for being in the current state, how likely it is to transition to other states, and the utility of those other states [3].

The equation looks like this :

$$U(S) = R(S) + \gamma \times \max_{a \in \{\text{ACTIONS}\}} \sum_{S'} P(S'|S,a) \times U(S')$$

In this equation:

U(S) is the utility of the current state.
R(S) is the reward for current state.
γ is the discount factor.
P(S'|S,a) is the probability of transitioning to state S' from the state S by taking action a.
U(S') is the utility of the next state.

The great thing about Value Iteration is that it can handle situations where the transitions between states and the rewards for taking actions are uncertain or random. However, one downside is that it assumes the agent knows everything about how the environment works.

For our maze search problem, we have two options for setting the transition probabilities:

1. Stochastic: This means the agent is more likely to move in certain directions based on probabilities we set.
2. Deterministic: This means the agent explores all directions equally.

We set the transition probabilities this way because the start and goal nodes are positioned in specific ways within the maze.

To solve the MDP, we follow a plan inspired by a book. We set the reward for reaching the goal as the highest, and the reward for being in other states is set lower. The discount factor is used to adjust how much we care about future rewards compared to immediate rewards. A higher discount factor means we care more about future rewards.

The goal of setting a higher discount factor is to help the algorithm converge faster. The algorithm keeps updating the utility of each state until it converges to the best solution.

The figure below (Figure 5) shows the path of an agent from the start node (10, 10) to the goal node (1, 1) in a 10x10 maze.

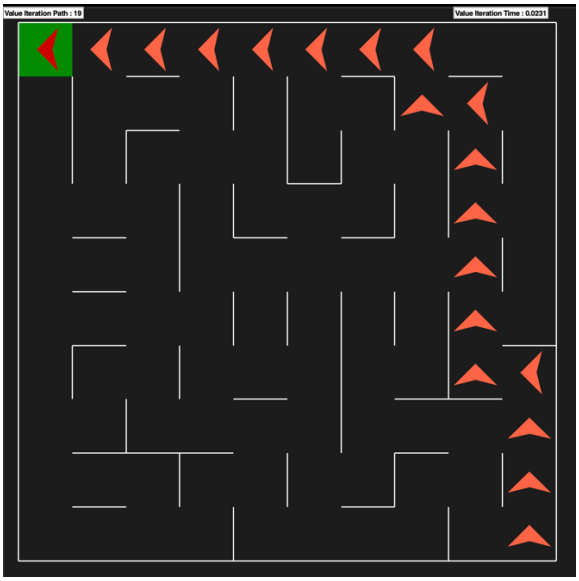

Figure 5 : Value Iteration on 10x10 maze

II POLICY ITERATIONS

In Policy Iteration, the initial step involves policy evaluation, akin to value iteration but with fixed policies, achieved by solving recursive Bellman equations. Subsequently, the policy is improved through iterative processes: first by evaluating it and then enhancing it through a greedy approach, utilizing one-step lookahead based on the utility of the next state.

The function employed to execute this policy improvement, as described in the pseudo-code, is as follows:

$$\pi[s] \leftarrow argmax\_a \sum s' P(s'|s,a)U[s']$$

The policy improvement occurs through iterative steps, halting when the improvement yields no substantial enhancement in utilities. Two approaches exist for defining transition probabilities: Stochastic and Deterministic.

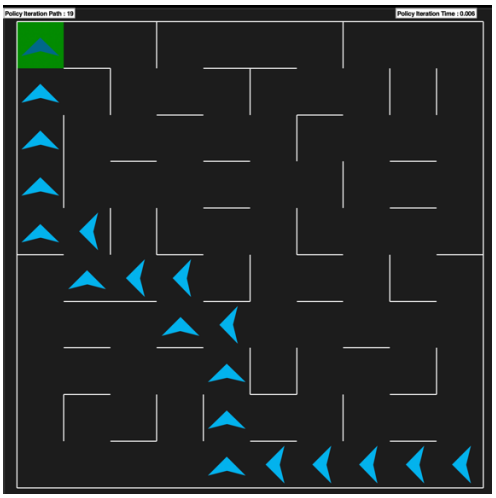

Figure 6 : Policy Iterations on 10x10 maze

4. Results

I. Comparing Performance Of Search Algorithms (DFS vs BFS vs A*)

In this section, I tested the search algorithms on many different mazes, each of different sizes, to compare the performance of each algorithm. As a metric, I have used the number of nodes in the path taken by an algorithm and the time taken by each algorithm to reach the goal state to judge the performance of the algorithms. The algorithms were tested over the following mazes: 5x5, 10x10, 20x20, 30x30, and 50x50.

The tables below illustrates how well the search algorithms performed across different mazes.

Maze Size	Time Taken (in Seconds)	Number of Steps
5x5	0.0000	9
10x10	0.0001	39
20x20	0.0011	83
30x30	0.0047	191
50x50	0.0461	607

Table 1 : Shows the performance of DFS algorithm across different maze size

Maze Size	Time Taken (in Seconds)	Number of Steps
5x5	0.0000	21
10x10	0.0002	49
20x20	0.0019	167

30x30	0.0090	303
50x50	0.0659	711

Table 2 : Shows the performance of BFS algorithm across different maze size

Maze Size	Time Taken (in Seconds)	Number of Steps
5x5	0.0001	15
10x10	0.0002	81
20x20	0.0005	105
30x30	0.0014	217
50x50	0.0051	1019

Table 3 : Shows the performance of A algorithm across different maze size*

Upon examining the performance of Depth-First Search (DFS), Breadth-First Search (BFS), and A* algorithms across mazes of varying sizes, it becomes apparent that each algorithm offers distinct advantages and trade-offs. DFS consistently demonstrates the shortest execution time and requires the fewest steps to find a solution. This makes it particularly suitable for smaller mazes or situations where finding a solution quickly is paramount. On the other hand, BFS guarantees finding the shortest path but tends to consume more time and memory due to its exhaustive exploration of all possible paths. Despite its computational demands, BFS excels in scenarios where finding the optimal solution is critical and computational resources are abundant.

A* algorithm strikes a balance between efficiency and optimality by considering both the cost to reach a node and the estimated cost to reach the goal. While not as fast as DFS, nor as exhaustive as BFS, A* offers a reasonable compromise, making it a versatile choice for various maze sizes and scenarios where finding a good solution efficiently is desired. Ultimately, the selection of the algorithm should be based on factors such as maze size, computational resources, and the importance of finding an optimal solution. Each algorithm presents its own set of advantages and limitations, and the optimal choice depends on the specific requirements of the problem at hand.

II. Comparison of Performance of MDP Algorithms (Value Iteration vs Policy Iteration)

MDP algorithms' performance is evaluated using the same metrics as those employed for search algorithms, namely time taken (in seconds) and the number of steps. It's important to note that I utilized loopPercent = 100 and DETERMINISTIC = True for both the Value and Policy Iteration algorithms.

The tables below illustrates how well the MDPs algorithms (Value & Policy Iteration) performed across different mazes.

Maze Size	Time Taken (in Seconds)	Number of Steps
5x5	0.0126	9
10x10	0.0253	19
20x20	0.0589	39
30x30	0.1111	59
50x50	0.2379	99

Table 4 : Shows the performance of Value Iteration across different maze size

Maze Size	Time Taken (in Seconds)	Number of Steps
5x5	0.0021	9

10x10	0.005	19
20x20	0.011	39
30x30	0.0117	59
50x50	0.0229	99

Table 5 : Shows the performance of Policy Iteration across different maze size

The comparison of performance metrics between Value Iteration and Policy Iteration across various maze sizes reveals interesting insights. In terms of time taken, both algorithms exhibit similar trends, with longer times recorded for larger maze sizes. However, Value Iteration generally takes slightly longer than Policy Iteration across all maze sizes. This discrepancy in time could be attributed to the iterative nature of Value Iteration, which requires multiple sweeps over the state space to converge to the optimal values.

On the other hand, Policy Iteration involves alternating between policy evaluation and policy improvement steps, which may lead to more efficient convergence. Regarding the number of steps, both algorithms demonstrate consistent behavior, with an increase in the number of steps as the maze size grows. Interestingly, both algorithms require the same number of steps for the same maze size, indicating that the complexity of the maze does not significantly impact the number of steps taken by either algorithm. Overall, while both Value Iteration and Policy Iteration offer effective solutions to maze navigation problems, Policy Iteration appears to perform slightly better in terms of time efficiency, making it a preferable choice for scenarios where computational resources are limited or time constraints are critical.

III. Comparison of Search Algorithms and MDP

Comparing the performance of search algorithms (DFS, BFS, A*) with MDP algorithms (Value Iteration, Policy Iteration) across different maze sizes reveals notable differences.

In terms of time taken, search algorithms generally exhibit faster performance than MDP algorithms. For smaller maze sizes (5x5 and 10x10), the time taken by search algorithms is significantly lower compared to MDP algorithms. However, as the maze size increases, the time taken by both types of algorithms increases, with MDP algorithms showing a more pronounced increase. This suggests that search algorithms scale better with larger maze sizes in terms of time complexity.

Regarding the number of steps, MDP algorithms tend to require fewer steps compared to search algorithms. This indicates that MDP algorithms are more efficient in terms of path planning, leading to shorter paths to reach the goal. However, this advantage diminishes as the maze size increases, with search algorithms sometimes outperforming MDP algorithms in terms of the number of steps for larger mazes.

Overall, search algorithms are more suitable for real-time applications where quick decision-making is crucial, especially for smaller mazes. On the other hand, MDP algorithms are better suited for scenarios where finding an optimal path with fewer steps is paramount, albeit at the cost of increased computational time, making them more suitable for offline planning or scenarios where computational resources are less constrained.

5. Conclusion

In conclusion, this assignment encompassed the implementation and comparison of various algorithms for maze-solving, including Depth-First Search (DFS), Breadth-First Search (BFS), and A* search, alongside Markov Decision Process (MDP) algorithms like Value Iteration and Policy Iteration. Through experimentation with different maze sizes, it became evident that search algorithms generally exhibit quicker performance, especially for smaller mazes, while MDP algorithms tend to yield shorter paths to

the goal. However, the advantage of MDP algorithms diminishes with larger maze sizes, where search algorithms may surpass them in terms of path length. Ultimately, the selection of an algorithm hinges on the specific needs of the application, with search algorithms favoured for real-time decision-making and MDP algorithms preferred for path optimization. This assignment provided valuable insights into the strengths and limitations of various algorithmic approaches for maze-solving.

Furthermore, the effectiveness of search algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), and A* search varied based on maze size and complexity. For smaller mazes, DFS and BFS algorithms typically performed better in terms of efficiency and path length due to their simplicity and minimal computational overhead. However, as maze size increased, A* search emerged as the superior choice, particularly regarding the number of steps required to reach the goal. This can be attributed to A* search's utilization of heuristic information, enabling it to navigate the maze more effectively and discover shorter paths. Overall, while DFS and BFS are well-suited for simpler mazes with manageable search spaces, A* search excels in larger and more intricate mazes where heuristic guidance is essential for optimal pathfinding.

7. References

[1] MAN1986. Pyamaze: A Python package for maze generation. https://github.com/MAN1986/pyamaze, Year Accessed.
[2] Ivana Dusparic. Lecture Notes, February 2024
[3] Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Pearson, 3 edition, 2010
[4] Learning Orbis. A-star a* search in python [python maze world- pyamaze], 2021.
2024
[5] Learning Orbis. Breadth first search (bfs) in python [python maze world- pyamaze], 2021.

APPENDIX:

1. Code to implement Depth First Search (DFS) algorithm.

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


def DFS(m):
    start = (m.rows, m.cols)
    explored = [start]
    frontier = [start]
    dfsPath = {}
    start_time = time.time()
    while len(frontier) > 0:
        currCell = frontier.pop()
        if currCell == (1, 1):
            break
        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    childCell = (currCell[0], currCell[1]+1)
                elif d == 'W':
                    childCell = (currCell[0], currCell[1]-1)
                elif d == 'S':
                    childCell = (currCell[0]+1, currCell[1])
                elif d == 'N':
                    childCell = (currCell[0]-1, currCell[1])
                if childCell in explored:
                    continue
                explored.append(childCell)
                frontier.append(childCell)
                dfsPath[childCell] = currCell
    end_time = time.time()
    fwdPath = {}
    cell = (1, 1)
    while cell != start:
        fwdPath[dfsPath[cell]] = cell
        cell = dfsPath[cell]
    return fwdPath, end_time - start_time


if __name__ == '__main__':
    print("\nPlease select Maze: \n")
    print("1. 5x5 Maze")
    print("2. 10x10 Maze")
    print("3. 20x20 Maze")
    print("4. 30x30 Maze")
    print("5. 50x50 Maze")
    print("6. Custom Maze")

    maze_choice = input("\nEnter your choice : ")

    if maze_choice == '1':
        m = maze(5, 5)
    elif maze_choice == '2':
        m = maze(10, 10)
    elif maze_choice == '3':
        m = maze(20, 20)
    elif maze_choice == '4':
        m = maze(30, 30)
    elif maze_choice == '5':
        m = maze(50, 50)
    elif maze_choice == '6':
        rows = int(input("Enter number of rows for custom maze: "))
        cols = int(input("Enter number of columns for custom maze: "))
        m = maze(rows, cols)
    else:
```

```
    print("Invalid choice!")
    exit()

m.CreateMaze()
path, time_taken = DFS(m)  # Retrieve path and algorithm time
a = agent(m, footprints=True, shape='arrow')
m.tracePath({a: path})
l = textLabel(m, 'DFS Path Length', len(path)+1)
print("Time taken by DFS algorithm: {:.4f} seconds".format(time_taken))
m.run()
```

2. Code to Implement Breadth First Search (BFS) algorithm

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


def BFS(m):
    start = (m.rows, m.cols)
    frontier = [start]
    explored = [start]
    bfsPath = {}
    start_time = time.time()
    while len(frontier) > 0:
        currCell = frontier.pop(0)
        if currCell == (1, 1):
            break
        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    childCell = (currCell[0], currCell[1]+1)
                elif d == 'W':
                    childCell = (currCell[0], currCell[1]-1)
                elif d == 'N':
                    childCell = (currCell[0]-1, currCell[1])
                elif d == 'S':
                    childCell = (currCell[0]+1, currCell[1])
                if childCell in explored:
                    continue
                frontier.append(childCell)
                explored.append(childCell)
                bfsPath[childCell] = currCell
    end_time = time.time()
    fwdPath = {}
    cell = (1, 1)
    while cell != start:
        fwdPath[bfsPath[cell]] = cell
        cell = bfsPath[cell]
    return fwdPath, end_time - start_time


if __name__ == '__main__':
    print("\nPlease select Maze: \n")
    print("1. 5x5 Maze")
    print("2. 10x10 Maze")
    print("3. 20x20 Maze")
    print("4. 30x30 Maze")
    print("5. 50x50 Maze")
    print("6. Custom Maze")

    maze_choice = input("\nEnter your choice : ")

    if maze_choice == '1':
        m = maze(5, 5)
    elif maze_choice == '2':
        m = maze(10, 10)
    elif maze_choice == '3':
        m = maze(20, 20)
    elif maze_choice == '4':
        m = maze(30, 30)
```

```
elif maze_choice == '5':
    m = maze(50, 50)
elif maze_choice == '6':
    rows = int(input("Enter number of rows for custom maze: "))
    cols = int(input("Enter number of columns for custom maze: "))
    m = maze(rows, cols)
else:
    print("Invalid choice!")
    exit()

m.CreateMaze()
path, time_taken = BFS(m)
a = agent(m, footprints=True, shape='arrow')
m.tracePath({a: path})
l = textLabel(m, 'BFS Path Length', len(path)+1)
print("Time taken by BFS algorithm: {:.4f} seconds".format(time_taken))
m.run()
```

3. Code to Implement A Star Search Algorithm

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


def h(cell1, cell2):
    x1, y1 = cell1
    x2, y2 = cell2

    return abs(x1-x2) + abs(y1-y2)


def aStar(m):
    start = (m.rows, m.cols)
    g_score = {cell: float('inf') for cell in m.grid}
    g_score[start] = 0
    f_score = {cell: float('inf') for cell in m.grid}
    f_score[start] = h(start, (1, 1))

    open = PriorityQueue()
    open.put((h(start, (1, 1)), h(start, (1, 1)), start))
    aPath = {}
    start_time = time.time()
    while not open.empty():
        currCell = open.get()[2]
        if currCell == (1, 1):
            break
        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    childCell = (currCell[0], currCell[1]+1)
                if d == 'W':
                    childCell = (currCell[0], currCell[1]-1)
                if d == 'N':
                    childCell = (currCell[0]-1, currCell[1])
                if d == 'S':
                    childCell = (currCell[0]+1, currCell[1])

                temp_g_score = g_score[currCell]+1
                temp_f_score = temp_g_score+h(childCell, (1, 1))

                if temp_f_score < f_score[childCell]:
                    g_score[childCell] = temp_g_score
                    f_score[childCell] = temp_f_score
                    open.put((temp_f_score, h(childCell, (1, 1)), childCell))
                    aPath[childCell] = currCell
    end_time = time.time()
    fwdPath = {}
    cell = (1, 1)
    while cell != start:
```

```
        fwdPath[aPath[cell]] = cell
        cell = aPath[cell]
    return fwdPath, end_time - start_time


if __name__ == '__main__':
    print("\nPlease select Maze: \n")
    print("1. 5x5 Maze")
    print("2. 10x10 Maze")
    print("3. 20x20 Maze")
    print("4. 30x30 Maze")
    print("5. 50x50 Maze")
    print("6. Custom Maze")

    maze_choice = input("\nEnter your choice : ")

    if maze_choice == '1':
        m = maze(5, 5)
    elif maze_choice == '2':
        m = maze(10, 10)
    elif maze_choice == '3':
        m = maze(20, 20)
    elif maze_choice == '4':
        m = maze(30, 30)
    elif maze_choice == '5':
        m = maze(50, 50)
    elif maze_choice == '6':
        rows = int(input("Enter number of rows for custom maze: "))
        cols = int(input("Enter number of columns for custom maze: "))
        m = maze(rows, cols)
    else:
        print("Invalid choice!")
        exit()

    m.CreateMaze()
    path, time_taken = aStar(m)

    a = agent(m, footprints=True, shape='arrow')
    m.tracePath({a: path})
    l = textLabel(m, 'A Star Path Length', len(path)+1)
    print("Time taken by A* algorithm: {:.4f} seconds".format(time_taken))
    m.run()
```

4. Code to Implement Value Iteration

```
class ValueIteration(MarkovDecisionProcess):
    def __init__(self, m=None, goal=None, isDeterministic=True):

        super().__init__(m, goal, isDeterministic)

        self._reward = -4  # LIVING REWARD
        self._max_error = 10 ** (-3)

        self.target = [self.goal]
        self.values = {state: 0 for state in self.actions.keys()}
        self.values[self.target[0]] = 1

        self.algoPath = {}

        self.explored = []

        self.mainTime = 0

    def get_maxDelta(self, delta, utilityMax, state):
        return max(delta, abs(utilityMax - self.values[state]))

    def calculate_valueIteration(self):
        start = time.time()
        while True:
            delta = 0
```

```
            for state in self.actions.keys():
                if state == self.target[0]:
                    continue

                utilityMax = float("-inf")

                for action, prob in self.actions[state].items():
                    for direction in action:
                        if self.m.maze_map[state][direction]:
                            childCell = self.move(state, direction)
                    reward = self._reward
                    if childCell == self.target[0]:
                        reward = 10000
                    utility = 0

                    utility += super().calculate_ValueIterationUtility(prob,
                                                reward, childCell, self.values)

                    if utility > utilityMax:
                        utilityMax = utility

                delta = self.get_maxDelta(delta, utilityMax, state)

                self.values[state] = utilityMax

            if delta < self._max_error:
                break
        end = time.time()
        self.mainTime = end-start

    def create_searchPath(self, currNode):
        start = time.time()
        node = currNode

        while True:
            selectedNode = None
            selectedNodeVal = None
            if node == self.target[0]:
                break

            for direction in 'ENWS':
                if self.m.maze_map[node][direction] and self.move(node, direction) not in self.explored:
                    traverseDirection = self.move(node, direction)
                    if traverseDirection == self.target[0]:
                        selectedNode = traverseDirection
                        break
                    if selectedNodeVal is None:
                        selectedNode = traverseDirection
                        selectedNodeVal = self.values[selectedNode]
                    else:
                        tempNode = traverseDirection
                        if selectedNodeVal < self.values[tempNode]:
                            selectedNode = tempNode
                            selectedNodeVal = self.values[tempNode]

            self.explored.append(node)
            self.algoPath[node] = selectedNode
            node = selectedNode
        end = time.time()
        self.mainTime = end-start
        return self.algoPath, self.mainTime
```

5. Code to Implement Policy Iteration

```
class PolicyIteration(MarkovDecisionProcess):
    def __init__(self, m=None, goal=None, isDeterministic=True):

        super().__init__(m, goal, isDeterministic)

        self.target = [self.goal]
```

```
        self.values = {state: 0 for state in self.actions.keys()}
        self.values[self.target[0]] = pow(10, 7)

        self.policyValues = {s: random.choice(
            'N') for s in self.actions.keys()}

        # LIVING REWARD
        self._reward = {state: -40 for state in self.actions.keys()}
        self._reward[self.target[0]] = pow(10, 8)

        self.algoPath = {}
        self.mainTime = 0

    def calculate_policyIteration(self):
        start = time.time()
        policyTrigger = True
        while policyTrigger:
            policyTrigger = False
            valueTrigger = True

            while valueTrigger:
                valueTrigger = False

                for state in self.actions.keys():
                    if state == self.target[0]:
                        continue

                    utilityMax = float('-infinity')
                    actionMax = None

                    for action, prob in self.actions[state].items():
                        for direction in action:
                            if self.m.maze_map[state][direction]:
                                childNode = self.move(state, direction)

                        utility = super().calculate_PolicyIterationUtility(prob, self._reward, state, childNode,
                                                                            self.values)

                        if utility > utilityMax:
                            utilityMax = utility
                            actionMax = action

                        self.policyValues[state] = actionMax
                        self.values[state] = utilityMax

                        if self.policyValues[state] != actionMax:
                            policyTrigger = True
                            self.policyValues[state] = actionMax

        end = time.time()
        self.mainTime = end-start
```