Artificial Intelligence Assignment 1 - Maze Solvers

Arnav Bhattacharya
Student ID: 22307812

Abstract—**This document analyses and summarizes the performance of different maze-solving algorithms, including Search Algorithms - Breadth First Search, Depth First Search and A-Star as well as Markov's Decision Process Algorithms using - Value Iteration and Policy Iteration.**

Index Terms—**Maze Solver, Markov's Decision Process, Value Iteration, Policy Iteration, Search Algorithms, Breadth First Search, Depth First Search, A-Star**

I. Introduction

Maze solving is a well-known problem in the field of computer science and artificial intelligence. The goal is to find a path from a starting point to a destination point in a maze, which may contain walls or obstacles. The problem can be solved using various search algorithms, including breadth-first search (BFS), depth-first search (DFS), A* search, and Markov Decision Process (MDP) algorithms such as Value Iteration and Policy Iteration.

In this performance analysis document, I aim to analyze and compare the performance of these algorithms in solving randomly generated mazes of different sizes. We will evaluate the algorithms based on various criteria, such as the time taken to find a solution, the number of nodes expanded(for search algorithms only), the memory consumption, the optimality of the solution and the scalability of the algorithm.

I ran the algorithms on different mazes of the same size multiple times and then analysed them.

II. Maze Implementation

For generating the random mazes of different sizes, I am using an open-source package named 'pyamaze' [1]. The link to the official repository and the post in the official Python Distributor is mentioned in the references below. I am generating the mazes by using the function `CreateMaze()`. We pass the `maze_object` which has the number of rows and columns of the maze to be generated. On running the `CreateMaze()` function with the parameter `loopPercent=50`, it creates a maze with multiple paths. The `loopPercentage` value is directly proportional to the number of multiple paths to be present in the generated maze. We can save the generated maze into a `csv` file. While development, this feature was used to generate the maze for one algorithm and use this saved maze to test the other algorithms.

III. Algorithm Explanation

A. Search Algorithms

1) Breadth First Search(BFS) Algorithm:: A My code implements the breadth-first search algorithm to find the shortest path from the starting point (maze.rows, maze.cols) to the ending point (1, 1) in a maze represented by the maze object.

The algorithm works as follows:

1) Initialize the starting cell as the current cell and add it to the `next_cell` list and the explored list.
2) While the `next_cell` list is not empty:
    a) Pop the first cell from the `next_cell` list and make it the `current_cell`.
    b) If the `current_cell` is the ending cell (1, 1), exit the loop.
    c) Otherwise, for each neighbouring cell that can be reached from the `current_cell`, check if it has already been explored. If it has not been explored, add it to the `next_cell` list, add it to the explored list, and record the path to it in the `bfs_path` dictionary.
    d) Append the `current_cell` to the `search_space` list.
3) Trace back the path from the ending cell to the starting cell using the `bfs_path` dictionary and create a `forward_path` dictionary that stores the path from the starting cell to the ending cell.
4) Return the `search_space` list and the `forward_path` dictionary.

In Summary, the algorithm explores all the cells in the maze in a breadth-first manner, and it stores the path from each explored cell to the starting cell in a dictionary. This enables the algorithm to trace back the shortest path from the ending cell to the starting cell.

2) Depth First Search(DFS) Algorithm:: B My code implements the depth-first search algorithm to find a path from the starting point (maze.rows, maze.cols) to the ending point (1, 1) in a maze represented by the maze object.

The algorithm works as follows:

1) Initialize the starting cell as the current cell and add it to the explored list and the `next_cell` list.
2) While the `next_cell` list is not empty:
    a) Pop the last cell from the `next_cell` list and make it the `current_cell`.
    b) If the `current_cell` is the ending cell (1, 1), exit the loop.
    c) Otherwise, for each neighbouring cell that can be reached from the `current_cell`, check if it has already been explored. If it has not been explored, add it to the explored list, add

it to the `next_cell` list, record the path to it in the `dfs_path` dictionary, and append the `current_cell` to the `search_space` list.

3) Trace back the path from the ending cell to the starting cell using the `dfs_path` dictionary and create a `forward_path` dictionary that stores the path from the starting cell to the ending cell.
4) Return the `search_space` list and the `forward_path` dictionary.

In summary, the algorithm explores the cells in the maze in a depth-first manner, and it stores the path from each explored cell to the starting cell in a dictionary. This enables the algorithm to trace back the path from the ending cell to the starting cell. Unlike the breadth-first search algorithm, the depth-first search algorithm does not guarantee to find the shortest path.

3) A Algorithm:* C The A* algorithm starts by defining a heuristic function that takes two cells as arguments and returns the Manhattan distance between them. I tested the algorithm using Euclidean distance but the efficiency of the algorithm increased by a huge margin on using the Manhattan distance.

The algorithm initializes two dictionaries `g_score` and `f_score`, which represent the cost to reach a cell from the starting cell and the cost to reach the goal from a cell respectively.

The algorithm initializes the search space with the starting cell and creates a priority queue with the starting cell as the only element. The priority queue is used to keep track of the cells that have been visited but not explored yet.

The algorithm enters a loop where it pops the cell with the lowest f-score from the priority queue and checks if it is the goal cell. If it is the goal cell, the algorithm terminates and returns the search space and the path.

If it is not the goal cell, the algorithm checks each neighbouring cell of the current cell and calculates the tentative g-score and f-score for each neighbour. If the tentative f-score is lower than the f-score of the neighbour, the g-score and f-score are updated, and the neighbour is added to the priority queue with its f-score as the priority value. The algorithm also adds the current cell to the search space and updates the path dictionary with the current cell as the parent of the neighbour cell.

Finally, the algorithm backtracks from the goal cell to the starting cell to find the optimal path and returns the search space and the optimal path.

The key difference between A* and the other search algorithms is that it uses the heuristic function to estimate the distance from a cell to the goal and guides the search towards the goal. This makes A* more efficient than other search algorithms, especially when the search space is large.

B. Markov's Decision Process Algorithms:

1) Value Iteration: : D Markov Decision Process (MDP) is a mathematical framework used to model decision-making problems where outcomes are partly random and partly under the control of a decision-maker. It consists of a set of states, actions, transition probabilities, and rewards. Value iteration is a popular algorithm used to solve MDP problems. It is an iterative algorithm that computes the optimal value function for each state.

Value Iteration follows the following equation:

$$V(s_t) = \max_{a_t} \left\{ R(s_t) + \gamma \sum_{s_t+1} P(s_{t+1}|s_t, a_t) V(s_{t+1}) \right\} \quad (1)$$

Where $V(s_t)$ is the state-value function for state s_t, which is the expected long-term reward of being in state s_t assuming an optimal policy is followed. $\max_{a_t}$ is the maximum over all possible actions a_t that can be taken in state s_t, representing the optimal action to take. $R(s_t)$ is the reward obtained in state s_t when action a_t is taken. γ is the discount factor, a constant value between 0 and 1 that determines the relative importance of future rewards versus immediate rewards. A discount factor of 0 means that only immediate rewards are considered, while a discount factor of 1 means that all future rewards are considered equally important as immediate rewards. $\sum_{s_{t+1}} P(s_{t+1}|s_t, a_t)$ is the sum of the probabilities of all possible next states s_{t+1}, given that action a_t is taken in state s_t, weighted by their transition probabilities $P(s_{t+1}|s_t, a_t)$. $V(s_{t+1})$ the value of the next state s_{t+1}, which is the expected long-term reward of being in state s_{t+1} assuming an optimal policy is followed.

My algorithm uses value iteration to determine the optimum strategy for a particular labyrinth. The labyrinth is depicted as a grid in which each cell represents a state. The algorithm attempts to locate the most efficient route from the bottom right corner of the labyrinth to the top left corner. Initially, the algorithm determines the threshold and decay hyperparameters. It then gives rewards to each of the maze's states and defines them as a list of cells. The rewards are specified as -1 for all states with the exception of the upper left corner, which gets a payout of 1000. Using the maze map, the algorithm determines the available actions for each state. It generates an action dictionary in which each state corresponds to a list of potential actions. Each state's starting policy is specified as a random action. The initial value function is also specified as -1 for all states save the upper-left corner, which has the value 10000. The algorithm then enters the main loop, which repeats until the change in the value function is less than the predetermined threshold. The algorithm assesses the value of each action for each state and picks the action with the greatest value. The chosen action becomes the new policy for that state, and the state's new value is updated depending on the specified action and transition probabilities. The best policy is returned as a list of states indicating the route from the bottom right corner to the top left corner. This code implements the value iteration technique to determine the ideal maze policy.

2) Policy Iteration: : E Markov Decision Process (MDP) is a mathematical framework for decision-making in a stochastic environment, where the outcome depends on both the current state and the action taken. It involves defining a set of states, actions, transition probabilities, and rewards. The goal is to find a policy that maximizes the expected cumulative reward.

In my code, the MDP is represented by the maze environment, which contains a set of states (`maze_map`), actions (`get_actions`), and rewards (rewards). The goal is to find the optimal policy for navigating the maze, which is done through the process of policy iteration.

Policy iteration involves two steps: policy evaluation and policy improvement. In policy evaluation, the value function (V) for each state is calculated for a given policy. The value function represents the expected cumulative reward starting from that state and following the given policy. This is done by iterating over all states and actions, calculating the expected reward for each action, and updating the value function using the Bellman equation:

$$v(s) = r(s) + \gamma \sum_{s'} p(s'|s, a = \Pi(s))v(s') \tag{2}$$

where $v(s)$ is the state-value function for state s, which is the expected return starting from state s.$r(s)$ is the immediate reward obtained when transitioning from state s to the next state by taking action a. γ is the discount factor, a constant value between 0 and 1 that determines the relative importance of future rewards versus immediate rewards. $\sum_{s'} p(s'|s, a = \Pi(s))$ is the sum of the expected values of all possible successor states s', weighted by their transition probabilities $p(s'|s, a = \Pi(s))$.

This is the predicted value of the subsequent state, given the present state and the policy, indicated by $\Pi(s)$. $v(s')$ is the value of the subsequent state s', which represents the anticipated return beginning with s'.

In policy enhancement, the policy is modified such that it is greedy with regard to the current value function. This is accomplished by repeatedly traversing all states and actions, computing the anticipated reward for each action, and selecting the action with the highest expected reward. If the new policy is identical to the previous policy, the algorithm has reached the optimum policy. Following the greedy policy beginning at the target state (1,1) and moving backwards to the start state yields the optimum route through the labyrinth as the final output of the code (maze.rows, maze.cols). Returns the ideal route as a list of states (`policy_path`).

IV. Performance Analysis

NOTE: The tables below consist of the average values of ten(10) iterations for a given maze size. This means that the respective algorithms are executed on ten(10) different mazes of the given size and then their individual values are averaged. Also, the start cell is set at (0,0)th cell and the goal cell is set at (row, column)th cell.

A. Search Algorithms

After executing the codes ten times per maze size, the mazes are generated at random and all of the Search algorithms for finding a path from the start cell to the goal cell are applied in the same maze generated. The results are tabulated below:

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.00376859	58.29960938	19.2	97.7
Depth First Search	0.00088629	58.29960938	23	44.6
A Star	0.00233239	58.30117188	22.4	26.1

TABLE I
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 10X10

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.0499553	60.50585938	43.6	99.975
Depth First Search	0.00541979	60.50585938	77.2	30.55
A Star	0.00611274	60.5078125	48.8	14.075

TABLE II
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 20X20

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.24475388	64.7453125	62.8	99.98
Depth First Search	0.00707387	64.7453125	92.8	17.4
A Star	0.01099324	64.76914063	78	10.03

TABLE III
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 30X30

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.81335739	71.18125	84.6	99.8875
Depth First Search	0.03202237	71.18203125	116.2	17.5625
A Star	0.01900944	71.2375	102	7.53125

TABLE IV
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 40X40

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	1.80624227	78.59648438	107	99.992
Depth First Search	0.07578138	78.59765625	197.4	19.164
A Star	0.0231798	78.68632813	126.4	5.86

TABLE V
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 50X50

Summary: As observed from the above tables(Table I-V), we can note the following observations:

1) **Time Consumption:** Breadth First Search algorithm takes the maximum amount of time to find a path, followed by the Depth First Search algorithm. The A-Star algorithm takes the least amount of time to find a

path.

2) **Memory Consumption:** Both Breadth First Search and Depth First Search algorithms consume almost the same memory to find a path. While the A-Star algorithm consumes slightly more memory than the other two algorithms.
3) **Length of Path:** Depth-First Search Algorithm finds the path of the longest length when compared to the other two search algorithms. Whereas the Breadth-First Search algorithm finds the path of the least length.
4) **Nodes Expanded:** Breadth-First Search Algorithm visits almost all of the nodes or cells in the maze and then finds the optimal path. Whereas A-Star visits the least percentage of cells to find a path.

These observations hold true when executing these algorithms for a given maze size.

Another observation worthy of note is that on increasing the size of the maze, the nodes covered by the A-Star Search algorithm reduce drastically.

B. Markov's Decision Process

After executing the codes ten times per maze size, the mazes are generated at random and both of the MDP algorithms for finding a path from the start cell to the goal cell are applied in the same maze generated. The results are tabulated below:

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)
Value Iteration	0.14302833	67.64140625	21
Policy Iteration	0.22285986	67.64453125	21

TABLE VI
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 10X10

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)
Value Iteration	0.32067546	68.862890625	32.2
Policy Iteration	0.50588228	68.89765625	32.2

TABLE VII
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 15X15

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)
Value Iteration	0.46908884	69.47265625	38.8
Policy Iteration	0.82467698	69.48125	38.8

TABLE VIII
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 15X20

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)
Value Iteration	0.61101367	69.915625	43.4
Policy Iteration	1.10592429	69.9390625	43.4

TABLE IX
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 20X20

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)
Value Iteration	0.93764074	71.759765625	53.2
Policy Iteration	1.9200653	71.81953125	53.2

TABLE X
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 25X25

Point to note: When the size of the mazes was more than 25x25, my system was unable to process all the data and was throwing heap errors for Value and Policy Iteration algorithms. This led to my system hanging and becoming unresponsive. Hence the comparison is performed on mazes of size smaller than 26x26. **Summary:** As observed from the above tables(Table VI-X), we can note the following observations:

1) **Time Consumption:** Value Iteration takes significantly lesser time to find the path when compared to Policy Iteration.
2) **Memory Consumption:** Policy Iteration consumes more memory while executing when compared to Value Iteration.
3) Both of the algorithms give the path of the same length

These observations hold true when executing these algorithms for a given maze size.

C. All Algorithms

After executing the codes ten times per maze size, the mazes are generated at random and all five of the algorithms(three search algorithms and two MDP algorithms) for finding a path from the start cell to the goal cell are applied in the same maze generated. The results are tabulated below:

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.00376859	58.29960938	19.2	97.7
Depth First Search	0.00088629	58.29960938	23	44.6
A Star	0.00233239	58.30117188	22.4	26.1
Value Iteration	0.14302833	67.64140625	21	-
Policy Iteration	0.22285986	67.64453125	21	-

TABLE XI
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 10X10

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.01611425	59.24140625	30.2	98.8
Depth First Search	0.00112752	59.24140625	37.2	24.7
A Star	0.00416037	59.24921875	37	19.8
Value Iteration	0.32067546	68.862890625	32.2	-
Policy Iteration	0.50588228	68.89765625	32.2	-

TABLE XII
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 15X15

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.0279557	59.971484375	36.8	99.1
Depth First Search	0.0030562	59.971484375	48.2	30.9
A Star	0.00521293	59.979296875	45	17.3
Value Iteration	0.46908884	69.47265625	38.8	-
Policy Iteration	0.82467698	69.48125	38.8	-

TABLE XIII
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 15X20

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.0499553	60.50585938	43.6	99.975
Depth First Search	0.00541979	60.50585938	77.2	30.55
A Star	0.00611274	60.5078125	48.8	14.075
Value Iteration	0.61101367	69.915625	43.4	-
Policy Iteration	1.10592429	69.9390625	43.4	-

TABLE XIV
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 20X20

	Time Complexity(s)	Memory Complexity(MiB)	Length of Path(Num)	Nodes Expanded(%)
Breadth First Search	0.1163742	62.391	53	99.98
Depth First Search	0.0075409	62.39023	88.2	24.74
A Star	0.0090898	62.4	66.8	13.42
Value Iteration	0.93764074	71.75977	53.2	-
Policy Iteration	1.9200653	71.8195	53.2	-

TABLE XV
AVERAGE VALUES OF 10 ITERATIONS FOR MAZES OF SIZE 25X25

NOTE: *The percentage of nodes expanded for Value and Policy Iteration is left empty for all the above tables as both of these algorithms visit all the cells in every iteration and calculate their respective values on each iteration until either of the value becomes smaller than the threshold value.*

Summary: As observed from the above tables(Table XI-XV), we can note the following observations:

1) **Time Consumption:** A-Star Algorithm takes the least amount of time to find a path whereas Policy Iteration takes the maximum amount of time.
2) **Memory Consumption:** There is an insignificant difference in the memory consumed by Breadth First Search and Depth First Search Algorithms. These two consume the least memory when compared to the rest. Whereas, Policy Iteration consumes the maximum amount of memory for finding a path.
3) **Length of Path:** Breadth First Search Algorithm usually gives the shortest path when compared to the other algorithms.

These observations hold true when executing these algorithms for a given maze size.

Even upon increasing the size of the maze, we see that the above observations hold true.

V. EVALUATION

As per our criteria of judgement mentioned before, here are the results of our observations for all the algorithms:

1) **Breadth First Search:** Since this algorithm visits almost all the nodes or cells in a maze and then finds the best path from the start to the goal cell, the time and memory consumed are quite high when compared to other search algorithms. But when compared with MDP algorithms, it definitely takes lesser time to provide us with the path. So, upon increasing the maze size, the time consumption will increase. Also, it provides us with the path of the smallest length when compared to other search and MDP algorithms. Even though this algorithm provides us with the smallest path length when compared to the other algorithms in the discussion, I believe it will consume significantly higher memory and time when the maze sizes are scaled up. Thus, it is not efficient for maze solving for mazes with larger sizes.
2) **Depth First Search:** Depth First Search first checks for the goal node depth-wise first and if it is not found, then it checks for the lateral nodes. So, for mazes where the start and goal nodes are present laterally, the algorithm would naturally have more time and memory complexity but still should be lesser than the Breadth-First Search algorithm. As observed from the tables above, this algorithm does not ensure that the most optimal path is provided. The number of nodes expanded is definitely lesser than the Breadth-First Search algorithm but is more than the A-Star algorithm. The MDP algorithms are not taken into consideration while evaluating the number of nodes expanded as they have multiple iterations and in each iteration, they visit every cell. The Depth First Search algorithm would not give the most optimal path from the start to the end cell in a maze but is surely efficient for pathfinding when the sizes of the mazes are scaled up.
3) **A-Star:** The A* algorithm starts by defining a heuristic function that takes two cells as arguments and returns the Manhattan distance between them. Based on the sum of the cell value and the heuristic value, the next cell to be explored is chosen by calculating the minimum of all the available options. This algorithm consumes the least time and memory for finding a path from the start to the end cell. Also, the number of nodes expanded is the least when compared to the other algorithms. The number of cells in the path from the start to the goal cell is larger than the Breadth-First Search, Value Iteration and Policy Iteration algorithm, but it is shorter than the

Depth First Search algorithm. Even though the A-Star algorithm does not provide the most optimal path from the start to the end cell, it does better than the Depth First Search algorithm. This algorithm is very efficient even when the size of the maze is scaled up. I tried the algorithm with a maze of size 50x50, and still, it was the most efficient one.

4) **Value Iteration:** The basic idea behind value iteration is to estimate the expected total reward the agent will receive by following a certain policy. By iterating over the table and updating the values of each state based on the values of its neighbouring states, we can gradually converge on the optimal policy for the agent. Since all the cells are visited in every iteration and the expected reward is calculated for each of them, the time and memory complexity is quite high when compared to the search algorithms, but they are lesser when compared to Policy Iteration. From the above tables, it is clearly visible that the length of the path found from the start to the goal cell is larger when compared to the Breadth First Search algorithm, but it is smaller when compared to the other algorithms. For mazes of size larger than 25x25, my system was unable to find a solution due to memory constraints and ended up becoming unresponsive. It is indeed scalable, but the system running the algorithm needs to be well-equipped.
5) **Policy Iteration:** First, we start with an initial policy which tells the agent what action to take in each state of the environment. Then we evaluate this policy by simulating the agent's actions in the environment and calculating a value function for each state. The value function represents the expected amount of reward the agent will receive if it starts in that state and follows the policy. Next, we improve the policy by making it greedy with respect to the value function. In other words, we update the policy to always choose the action that leads to the state with the highest value. We continue the above two steps until the policy no longer changes. Due to this, the algorithm consumes a lot of time and memory. Since all the cells are visited in every iteration and the expected reward is calculated for each of them, the time and memory complexity is quite high when compared to the search algorithms. From the above tables, it is clearly visible that the length of the path found from the start to the goal cell is larger when compared to the Breadth First Search algorithm, but it is smaller when compared to the other algorithms. For mazes of size larger than 25x25, my system was unable to find a solution due to memory constraints and ended up becoming unresponsive. It is indeed scalable, but the system running the algorithm needs to be well-equipped.

References

[1] pyamaze python package: GitHub, PyPI

Appendix

Appendix A
Breadth First Search

```
def breadth_first_search(maze):
    start = (maze.rows, maze.cols)
    next_cell = [start]
    explored = [start]
    bfs_path = {}
    search_space = []
    while len(next_cell) > 0:
        current_cell = next_cell.pop(0)
        if current_cell == (1, 1):
            break
        for direction in 'ESNW':
            if maze.maze_map[current_cell]
                        [direction] == True:
                if direction == 'E':
                    child_cell =
                    (current_cell[0],
                    current_cell[1]+1)
                elif direction == 'W':
                    child_cell =
                    (current_cell[0],
                    current_cell[1]-1)
                elif direction == 'N':
                    child_cell =
                    (current_cell[0]-1,
                    current_cell[1])
                elif direction == 'S':
                    child_cell =
                    (current_cell[0]+1,
                    current_cell[1])
                if child_cell in explored:
                    continue
                next_cell.append(child_cell)
                explored.append(child_cell)
                bfs_path[child_cell] =
                current_cell
                search_space.append(child_cell)
    forward_path = {}
    cell = (1, 1)
    while cell != start:
        forward_path[bfs_path[cell]] = cell
        cell = bfs_path[cell]
    return search_space, forward_path
```

Appendix B
Depth First Search

```
def depth_first_search(maze):
start = (maze.rows, maze.cols)
explored = [start]
next_cell = [start]
dfs_path = {}
search_space = []
while len(next_cell) > 0:
        current_cell = next_cell.pop()
        if current_cell == (1, 1):
            break
        for direction in 'ESNW':
            if maze.maze_map[current_cell]
                            [direction] == True:
                if direction == 'E':
                    child_cell =
                    (current_cell[0],
                    current_cell[1]+1)
                elif direction == 'W':
                    child_cell =
                    (current_cell[0],
                    current_cell[1]-1)
                elif direction == 'S':
                    child_cell =
                    (current_cell[0]+1,
                    current_cell[1])
                elif direction == 'N':
                    child_cell =
                    (current_cell[0]-1,
                    current_cell[1])
                if child_cell in explored:
                    continue
                explored.append(child_cell)
                next_cell.append(child_cell)
                dfs_path[child_cell] =
                current_cell
                search_space.append
                (current_cell)

    forward_path = {}
    cell = (1, 1)
    while cell != start:
        forward_path[dfs_path[cell]] = cell
        cell = dfs_path[cell]
    return search_space, forward_path
```

Appendix C
A-Star

```
def heuristic(cell1,cell2):
  x1,y1=cell1
  x2,y2=cell2
  return abs(x1-x2)+abs(y1-y2)

def a_star(maze):
  start=(maze.rows,maze.cols)
  g_score={cell:float('inf') for cell
            in maze.grid}
  g_score[start]=0
  f_score={cell:float('inf') for cell
            in maze.grid}
  f_score[start]=heuristic(start,(1,1))
  search_space=[start]

  open=PriorityQueue()
  open.put((heuristic(start,(1,1)),
  heuristic(start,(1,1)),start))
  a_path={}
  while not open.empty():
    current_cell=open.get()[2]
    search_space.append(current_cell)
    if current_cell==(1,1):
      break
    for direction in 'ESNW':
      if maze.maze_map[current_cell]
                     [direction]==True:
        if direction=='E':
          child_cell=(current_cell[0],
            current_cell[1]+1)
        elif direction=='W':
          child_cell=(current_cell[0],
            current_cell[1]-1)
        elif direction=='N':
          child_cell=(current_cell[0]-1,
            current_cell[1])
        elif direction=='S':
          child_cell=(current_cell[0]+1,
            current_cell[1])
        temp_g_score=g_score[current_cell]
        temp_f_score=temp_g_score
            +heuristic(child_cell,(1,1))
        if temp_f_score<f_score[child_cell]:
          g_score[child_cell]=temp_g_score
          f_score[child_cell]=temp_f_score
          open.put((temp_f_score,heuristic(
          child_cell,(1,1)),child_cell)
          )
          a_path[child_cell]=current_cell
  forward_path={}
  cell=(1,1)
  while cell!=start:
    forward_path[a_path[cell]]=cell
    cell=a_path[cell]
  return search_space,forward_path
```

Appendix D
Value Iteration

```
def get_actions(maze,x,y):
    actions=maze.maze_map[(x,y)]
    s=[]
    for direction, bool in actions.items():
        if bool==1:
            s.append(direction)
    return s

def value_iteration(maze):
    #Hyperparameters
    threshold = 0.005
    # Threshold. if change is less than this
    # value then break.
    decay = 0.9    # decay
```

```
#Define all states
cell_list=list(maze.maze_map.keys())
#Define rewards for all states
rewards = {}
for i in cell_list:
    if i == (1,1):
        rewards[i] = 1000
    else:
        rewards[i] = -1
#Dictionnary of possible actions.
#We have two "end" states (1,2 and 2,2)
actions = {}
for i in range(1,maze.rows+1):
    for j in range(1,maze.cols+1):
        currCell = (i,j)
        actions[currCell] = ()
        for direction in 'ESNW':
            if maze.maze_map[currCell]
                        [direction]==True:
                actions[currCell]=
                    get_actions(maze,i,j)
#Define an initial policy
# From any cell, which is the
# best next cell to go to
policy={}
for action in actions.keys():
    policy[action] =
        np.random.choice(actions[action])
#Define initial value function
value={}
for cell in cell_list:
    if cell in actions.keys():
        value[cell] = -1
    if cell ==(1,1):
        value[cell]= 10000
iteration = 0
while True:
    delta = 0
    for cell in cell_list:
        old_v = value[cell]
        new_value = 0
        for action in actions[cell]:
            if action == 'N':
                nxt =
                    [cell[0]-1,
                    cell[1]]
            if action == 'S':
                nxt =
                    [cell[0]+1,
                    cell[1]]
            if action == 'W':
                nxt = [cell[0],
                cell[1]-1]
            if action == 'E':
                nxt =
                    [cell[0],
                    cell[1]+1]
            #Calculate the value
            v = rewards[cell]
                + (decay *
                value[tuple(nxt)])
            #Is this the best
            # action so far?
            # If so, keep it
            if v > new_value:
                new_value = v
                policy[cell] = action
        #Save the best of all
        #actions for the state
        value[cell] = new_value
        delta = max(delta,
        np.abs(old_v - value[cell]))
    #See if the loop should stop now
    if delta < threshold:
        break
    iteration += 1
# Get the greedy path
value_path = []
current_cell = (maze.rows,maze.cols)
value_path.append(current_cell)
while True:
    dir = policy[current_cell]
    if dir == 'N':
        current_cell =
            (current_cell[0]-1,
            current_cell[1])
    if dir == 'E':
        current_cell =
        (current_cell[0],
        current_cell[1]+1)
    if dir == 'S':
        current_cell =
        (current_cell[0]+1,
        current_cell[1])
    if dir == 'W':
        current_cell =
        (current_cell[0],
        current_cell[1]-1)
    value_path.append(current_cell)
    if current_cell == (1,1):
        break
return value_path
```

Appendix E
Policy Iteration

```
def get_actions(maze,x,y):
actions=maze.maze_map[(x,y)]
s=[]
for dir, bool in actions.items():
    if bool==1:
        s.append(dir)
return s
```

```
def policy_evaluation(policy, maze, V,
        gamma, theta):
    while True:
        delta = 0
        for s in maze.maze_map.keys():
            v = 0
            for a, a_prob in policy[s].items():
                nxt = get_next_state(maze, s, a)
                v += a_prob
                    * (maze.rewards.get
                            (nxt, 0)
                        + gamma * V[nxt])
            delta = max(delta, abs(v - V[s]))
            V[s] = v
        if delta < theta:
            break
def policy_improvement(maze, V, gamma):
    policy = {}
    for s in maze.maze_map.keys():
        policy[s] = {}
        for a in get_actions(maze, *s):
            policy[s][a] = 1.0
                / len(get_actions(maze, *s))
    while True:
        policy_evaluation(policy, maze,
            V, gamma, theta=0.001)
        policy_stable = True
        for s in maze.maze_map.keys():
            old_action = max(policy[s],
                key=policy[s].get)
            action_values = {}
            for a in get_actions(maze, *s):
                nxt =
                get_next_state(maze, s, a)
                action_values[a] =
                    maze.rewards.get(nxt, 0)
                        + gamma * V[nxt]
            best_action =
                max(action_values,
                    key=action_values.get)
            if old_action != best_action:
                policy_stable = False
            for a in policy[s]:
                policy[s][a] = 1
                    if a == best_action
                        else 0
        if policy_stable:
            break
    return policy
def get_next_state(maze, state, action):
    if action == 'N':
        nxt = (state[0]-1, state[1])
    elif action == 'S':
        nxt = (state[0]+1, state[1])
    elif action == 'W':
        nxt = (state[0], state[1]-1)
    elif action == 'E':
        nxt = (state[0], state[1]+1)
    else:
        nxt = state
    if nxt in maze.maze_map:
        return nxt
    else:
        return state
def policy_iteration(maze):
    # Hyperparameters
    gamma = 0.9
    # Define rewards for all states
    maze.rewards = {}
    for state in maze.maze_map:
        if state == (1, 1):
            maze.rewards[state] = 1000
        else:
            maze.rewards[state] = -1
    # Define initial value function
    V = {}
    for state in maze.maze_map:
        V[state] = 0
    # Policy iteration
    policy = policy_improvement(maze, V, gamma)
    # Get the greedy path
    policy_path = []
    state = (maze.rows, maze.cols)
    policy_path.append(state)
    while state != (1, 1):
        action = max(policy[state],
            key=policy[state].get)
        state =
            get_next_state(maze, state, action)
        policy_path.append(state)
    return policy_path
```