Artificial Intelligence I - Assignment 1
Maze search using search and MDP algorithms

Prathamesh Sai Sankar
5th year Integrated Computer Science
saisankp@tcd.ie
19314123

Abstract

This report analyzes the performance of various search and Markov Decision Process (MDP) algorithms in a maze search. The search algorithms used are depth-first search, breadth-first search and A*. The MDP algorithms used are policy iteration and value iteration. The comparison between the search algorithms, between the MDP algorithms, and between both can give us valuable insights into their underlying logic as well as the advantages and disadvantages of each.

1 Introduction

Algorithms can be used in a maze search to find a path of squares from a starting square to a target square, traversing across maze walls in this process. Some search algorithms that can be used include depth-first search, breadth-first search, and A*. Markov decision process (MDP) algorithms that can be used include policy iteration and value iteration. This report will analyze the performance of both search and MDP algorithms in a maze search consisting of various maze sizes that are randomly generated. To compare the algorithms, we can use metrics such as memory consumption, time taken and the number of squares required for the resulting path (for how optimal and cost effective it is). We can also use the number of squares for the search space for search algorithms in particular (for how efficiently it can find the resulting path).

2 Maze generator usage

To compare the search and MDP algorithms, I used an open source maze generator library in Python to generate mazes of various sizes using pyamaze [1] which can be found on GitHub. This library allows mazes to be created using the `CreateMaze()` method where we can pass 50 for the parameter `loopPercent` to have various paths in our maze (from a scale of 0 containing just one path from the starting square to target square to 100 containing as many loops and paths in the maze as possible). I implemented the algorithms so that they find a path from starting square (numberOfRowsInMaze, numberOfColumnsInMaze) i.e. the bottom right corner, to the ending square (1,1) for simplicity i.e. the top left corner. I kept this starting square and ending square for all algorithms to make a fair comparison between all of them.

3 Algorithm implementations

3.1 Depth-first search

My project implements depth-first search in `dfs.py` to find a path from the starting square (numberOfRowsInMaze, numberOfColumnsInMaze) to the ending square which is (1,1). The logic behind the algorithm is:

1. Begin with the starting square and add it to the lists `discovered_squares` and `subsequent_squares`.
2. Keep looping until there are no more squares in the `subsequent_squares` list.
3. While looping, slice the **last** element out of the list `subsequent_squares` and make it `present_square`.
4. Break out of the loop if we reached the ending square (1,1).
5. If we haven't reached the ending square yet, check all of the `present_square`'s neighbour squares to see if it has been discovered yet, adding them to the list of `discovered_squares` if not.
6. Append the `present_square` to a list tracking the search space called `maze_area_to_search` and store its path in the Python Dict `explored_squares`.
7. Once the loop is finished, use backtracking to complete the resulting path using `explored_squares`, so that we can return this path along with the search space.

The algorithm keeps track of the path from every new explored square to the starting square in a Python Dict called `explored_squares`. Using this, it backtracks to find the complete path from the starting square to the ending square to complete depth-first search which **may not be the shortest path**.

3.2 Breadth-first search

My project implements breadth first search in `bfs.py` to find the shortest path from the starting square (numberOfRowsInMaze, numberOfColumnsInMaze) to the ending square (1,1). The logic behind the algorithm is:

1. Begin with the starting square and add it to the lists `discovered_squares` and `subsequent_squares`.
2. Keep looping until there are no more squares in the `subsequent_squares` list.

3. While looping, slice the **first** element out of the list `subsequent_squares` and make it `present_square`.
4. Break out of the loop if we reached the ending square (1,1).
5. If we haven't reached the ending square yet, check all of `present_square`'s neighbour squares to see if it has been discovered yet, adding them to the `discovered_squares` list if not.
6. Append the `present_square` to a list tracking the search space called `maze_area_to_search` and store its path in the Python Dict `explored_squares`.
7. Once the loop is finished, use backtracking to complete the resulting path using `explored_squares`, so that we can return this path along with the search space.

The algorithm keeps track of the path from every new explored square to the starting square in a Python Dict called `explored_squares`. Using this, it backtracks to find the complete path from the starting square to the ending square to complete breadth-first search which to **find the shortest path**.

3.3 A*

My project implements A* in `a_star.py` to find a path from (numberOfRowsInMaze, numberOfColumnsInMaze) which is the starting square to (1,1) which is the ending square. It uses a heuristic to do so, and I used the manhattan distance as a heuristic to find the distance between two squares. I used this because it more accurately represents how an agent would move in a maze and does not underestimate the distance like Euclidean distance would (using straight lines). Taxicab geometry from the manhattan distance is like a zigzag which is similar to a maze. The resulting logic behind the algorithm is:

1. Add the starting square to a list tracking the search space and also to a priority queue `nodes_to_explore` to store squares that have not been explored yet.
2. Initialize a `g_score` Python Dict to track the cost from the start and a `f_score` Python Dict to track the cost from the start to the present square as well as the estimated cost from the present square to the goal.
3. Loop until we have no more nodes to explore.
4. While looping, get the square with the lowest f-score from the priority queue using `get()` and determine if it is the ending square, breaking out of the loop if it is.
5. If it is not the ending square, check all of the neighbours of the `present_square` to calculate intermediate g-scores and f-scores for each neighbour.
6. If the intermediate f-score is less than the previous f-score, we replace the f-score and g-score with the newer values, while also adding the neighbour to the priority queue `nodes_to_explore` using the new f-score as the key (that the priority queue uses to order them all).
7. While updating the scores, `explored_squares` is updated to link the `present_square` and neighbour, along with `present_square` being added to the search space.
8. Once the loop is finished, use backtracking to complete the resulting path using `explored_squares`, so that we can return this path along with the search space.

The use of the manhattan distance as the heuristic allows the A* algorithm to estimate the cost from the present square to the ending square and find the optimal path, which is different from the previous 2 algorithms (depth-first search and breadth-first search) that would be less efficient at scale.

3.4 Policy iteration

Policy iteration involves finding an optimal policy that is associated with a high reward for problems based around a Markov decision process (MDP). In this case, the MDP is a maze that has rewards that are achieved using actions (from the `valid_actions()` method) in states in `maze.maze_map` which are possible squares. Policy iteration consists of policy evaluation (where the value function for each state is calculated) and policy improvement (where the policy is adjusted in a greedy-first manner for the value function by iterating through all states and actions, calculating the reward for each action, and selecting the action with the highest expected reward).

The value function is the expected cumulative reward from a state using a policy. It is calculated by iterating through all states and actions to find the expected reward for each action as follows with the Bellman equation:

$$V(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s, a = \Pi(s)) \cdot V(s')$$

This equation[2] calculates the value function for a state $V(s)$ to get the expected reward from state s. The immediate reward for going from state s to the next state is $R(s)$. To establish a level of importance for rewards that are present versus in the future, we use a discount factor γ between 0 and 1, where 0 means immediate rewards are prioritized and 1 means future rewards have the same importance as immediate rewards. I chose 0.9 for the discount factor as it strikes a balance between immediate and long term rewards (for example, instead of 0.5 which would mean future rewards are discounted by 50%). This is multiplied by the sum of all expected values for all states that follow. When we complete policy improvement after this, we reach convergence if the resulting policy is the same as the previous policy. This optimal policy is used to begin at the state (numberOfRowsInMaze, numberOfColumnsInMaze) and go to the state (1,1) using an efficient route.

3.5 Value iteration

Value iteration involves finding the optimal value function for every state using an iterative process with the equation:

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

This equation[3] calculates the value function $V(s_t)$ for the state s_t. When an optimal policy is used, this represents the long term reward for being in that state. Finding the $\max_{a_t}$ gives the best action possible from the set of all actions a_t for state s_t. The discount factor γ between 0 and 1 is used to establish a level of importance for rewards that are present versus in the future. This is multiplied by the sum of the probabilities of all next states that can succeed the current state (i.e. s_{t+1}) that are weighted by their transition probabilities. This is multiplied by the value of the next state, representing the long term reward of being in the next state when an optimal policy is being used.

My project uses value iteration to find the best route from (numberOfRowsInMaze, numberOfColumnsInMaze) which is the starting square to (1,1) which is the ending square. The threshold and discount factor hyperparameters are set to 0.005 and 0.9. I chose 0.005 for the threshold because I wanted a precise solution that converges with little changes afterward (choosing a high threshold increases the risk that the algorithm converges at a non-optimal solution). I chose 0.9 for the decay as it strikes a balance between immediate and long term rewards (for example, instead of 0.5 which would mean future rewards are discounted by 50%). The rewards are set as -1 for non-target squares and 1000 for the target square which is (1,1). A Python Dict of available actions for each state is determined using the maze map which represents the list of possible actions.

The algorithm starts off by choosing a random action for the starting policy of each state, where the value function is 10000 for the target state and -1 for non-target states. This algorithm loops until the maximum change in the value function is less than the threshold 0.005. At each iteration, the algorithm calculates the value of every action for every state, choosing the action with the greatest value. Therefore, the selected action becomes the new policy for that state, resulting in a new value for that state. The policy that is most optimal is returned as a list of states to go from (numberOfRowsInMaze, numberOfColumnsInMaze) to (1,1).

4 Performance analysis

4.1 Between search algorithms

The search algorithms A*, depth-first search (DFS) and breadth-first search (BFS) were ran 15 times for mazes of 3 different sizes (10x10, 30x30 and 60x60). The algorithms start at (numberOfRowsInMaze, numberOfColumnsInMaze) to find the ending square at square (1,1). The metrics used are the maximum memory used (in MiB), the search space (number of squares), the resulting path (number of squares), and the time taken (in seconds). Memory and the search space show us how the algorithm has an impact on the computer's RAM. The length of the resulting path and the time taken shows how efficient the resulting route is, as well as how quick a computer can compute it. The results for different maze sizes are as follows:

	Max memory	Path length	Search space	Time
A*	17.62 MiB	21 squares	23 squares	0.042s
DFS	17.21 MiB	33 squares	55 squares	0.032s
BFS	17.15 MiB	19 squares	100 squares	0.009s

Table 1. 15 iterations of a 10x10 maze

	Max memory	Path length	Search space	Time
A*	20.93 MiB	83 squares	98 squares	0.019s
DFS	18.00 MiB	361 squares	751 squares	0.294s
BFS	17.93 MiB	67 squares	900 squares	0.478s

Table 2. 15 iterations of a 30x30 maze

	Max memory	Path length	Search space	Time
A*	27.66 MiB	163 squares	189 squares	0.042s
DFS	27.88 MiB	1239 squares	2997 squares	4.571s
BFS	29.74 MiB	137 squares	3600 squares	7.290s

Table 3. 15 iterations of a 60x60 maze

In terms of **memory**, breadth-first search uses the most memory as the maze size scales up. Depth-first search is very similar, but uses slightly less memory at scale. A* uses the least memory as the maze size scales up, but slightly more memory for mazes that are small compared to the other 2.

In terms of **path length**, depth-first search finds a path that is the longest (hence not optimal). A* is more optimal since it finds a path that needs less squares, but breadth-first search finds the path with the least squares (most optimal).

In terms of **search space**, breadth-first search has the most squares since it has to explore all squares at a given depth before moving to the next level. Depth-first search has a similarly high number of squares for its search space since it goes through deep depths before backtracking when it finds a dead end. The use of heuristics makes A* have the lowest search space, which makes a big impact as the maze size scales up.

In terms of **time**, we can see that the breadth-first search algorithm uses the most time to find a path as the size of the maze increases. Depth-first search takes less time, but it is not the quickest. A* takes the least amount of time, and this difference is very noticable as the maze size scales up.

4.2 Between MDP algorithms

The MDP algorithms policy iteration and value iteration were ran 15 times for mazes of 3 different sizes from (10x10, 30x30 and 60x60). The algorithms start at (numberOfRowsInMaze, numberOfColumnsInMaze) to find the ending square (1,1). The metrics used are the maximum memory used (in MiB), the resulting path (number of squares), and the time taken (in seconds). Memory shows us how the algorithm has an impact on the computer's RAM. The length of the resulting path and the time taken shows how efficient the resulting route is, as well as how quick a computer can compute it. The results for different maze sizes are as follows:

	Max memory	Path length	Time
Policy iteration	31.73 MiB	20 squares	0.469s
Value iteration	25.93 MiB	20 squares	0.213s

Table 4. 15 iterations of a 10x10 maze

	Max memory	Path length	Time
Policy iteration	38.32 MiB	62 squares	6.993s
Value iteration	31.81 MiB	62 squares	2.004s

Table 5. 15 iterations of a 30x30 maze

	Max memory	Path length	Time
Policy iteration	43.35 MiB	124 squares	52.075s
Value iteration	39.43 MiB	124 squares	25.032s

Table 6. 15 iterations of a 60x60 maze

In terms of **memory**, policy iteration requires the most memory. Value iteration requires less memory, and this becomes more evident as the size of the maze increases.

In terms of **path length**, policy iteration and value iteration have the same number of squares for their resulting paths. They both try to find the best policy and both result in a route that have the same number of squares.

In terms of **time**, policy iteration takes the most time. Value iteration takes significantly less time, and this becomes more evident as the size of the maze increases.

Interesting observation: Something I noticed was that the MDP algorithms policy iteration and value iteration both struggled with maze sizes over 30x30. Using 60x60 took almost 1 minute for policy iteration and I experienced my computer (M1 Macbook Pro with 8GB RAM) freezing at various times when I ran them both. I believe it would be infeasable to use them for any maze size that is higher than 60x60.

4.3 Between all algorithms

The search algorithms A*, depth-first search (DFS), breadth-first search (BFS) and MDP algorithms policy iteration (PI) and value iteration (VI) were all ran 15 times for mazes of 3 different sizes (15x15, 25x25 and 30x30). The algorithms start at (numberOfRowsInMaze, numberOfColumnsInMaze) to find the ending square (1,1). The metrics used are the maximum memory used (in MiB), the search space (number of squares, but only for search algorithms), the resulting path (number of squares), and the time taken (in seconds). Memory and the search space shows us how the algorithm has an impact on the computer's RAM. The length of the resulting path and the time taken shows how efficient the resulting route is, as well as how quick a computer can compute it. The results for different maze sizes are as follows:

	Max memory	Path length	Search space	Time
A*	30.18 MiB	37 squares	43 squares	0.008s
DFS	20.36 MiB	63 squares	103 squares	0.008s
BFS	19.35 MiB	31 squares	225 squares	0.036s
PI	19.50 MiB	32 squares	N/A	1.110s
VI	19.54 MiB	32 squares	N/A	0.490s

Table 7. 15 iterations of a 15x15 maze

	Max memory	Path length	Search space	Time
A*	30.21 MiB	57 squares	69 squares	0.013s
DFS	30.30 MiB	263 squares	514 squares	0.139s
BFS	30.29 MiB	51 squares	624 squares	0.235s
PI	30.52 MiB	52 squares	N/A	3.811s
VI	30.59 MiB	52 squares	N/A	1.416s

Table 8. 15 iterations of a 25x25 maze

	Max memory	Path length	Search space	Time
A*	34.96 MiB	87 squares	91 squares	0.019s
DFS	35.61 MiB	371 squares	955 squares	0.495s
BFS	35.70 MiB	75 squares	1225 squares	0.874s
PI	36.28 MiB	76 squares	N/A	9.564s
VI	35.12 MiB	76 squares	N/A	6.423s

Table 9. 15 iterations of a 35x35 maze

In terms of **memory**, policy iteration uses the most memory as the maze size scales up. Also, depth-first search and breadth first search have very similar memory usage at scale. Value iteration uses less memory than them, but A* uses the least memory at scale. A* uses the most memory at small maze sizes but at large maze sizes it uses the least memory.

In terms of **path length**, depth-first search has the highest path length (least optimal). A* has a lower path length than depth-first search. Policy iteration and value iteration are similar to breadth-first search in terms of path length, but slightly less optimal. Breadth-first search has the shortest path length (most optimal).

In terms of **time**, A* takes the least amount of time as the maze size scales up. Depth-first search and breadth-first search take more time but are relatively similar. Value iteration takes up significantly more time, but policy iteration takes the most amount of time to find a path from the start square to the target square.

5 Analysis of results

This section will summarize the results of running the search and MDP algorithms for different maze sizes. It will also explain why they occurred by discussing the underlying logic in each algorithm that caused these results.

For **depth-first search**, it uses less memory than breadth-first search but it is still relatively expensive memory-wise. This is because the algorithm explores deep depths before it backtracks if it hits a dead end. If the randomly generated maze has many branches that are deep, this algorithm would struggle to create a short path. It does not ensure that the resulting path is optimal with the least squares necessary. This is a fairly efficient algorithm as long as the absolute shortest path is not required and the maze does not have a lot of branches with deep paths leading to many dead ends.

For **breadth-first search**, it uses a lot of memory compared to other algorithms because it has to explore all the squares at a given depth before it can move up to the next square. It takes less time compared to MDP algorithms, and subsequently always results in the shortest path, but the memory consumed is the biggest downside which makes it inefficient. This algorithm would be viable if memory consumption is not an issue and the absolute shortest path is required.

For **A***, the use of a heuristic with the manhattan distance makes the algorithm use the least amount of memory and time. By choosing squares with the lowest heuristic value (lowest cost), the algorithm also does not need to have a bigger search space compared to depth-first search and breadth-first search which need to traverse a lot of squares needlessly to get to the target or policy iteration and value iteration which need to store value functions or policies for each state. The resulting path from A* is efficient and competitive with breadth-first search, while also being significantly better with time taken and memory used.

For **policy iteration**, it uses the most memory out of all algorithms because it uses an initial policy that is evaluated and improved by calculating the value function for each state. Using rewards, actions and states makes the algorithm find an efficient final path but the time taken is significantly higher compared to the other algorithms. For maze sizes that are bigger than 30x30, the program often froze since it required the storage of the value function and policy for each square (unlike value iteration which only stores the value) which takes a lot of memory. However, the resulting path is similar to the optimal path given by the breadth-first search algorithm which takes a similar amount of memory but significantly less time.

For **value iteration**, it has the 2nd highest use of memory because the estimation of the total reward by following a policy requires the storing and updating of values of each state until we converge since we have to visit every square in every iteration. We do not need to store a policy for every state (unlike policy iteration which does) so it does not use as much memory as policy iteration, but the path length is the same as policy iteration. The resulting path length is generally shorter than the 3 search algorithms but this comes at the cost of the time taken since the main focus of MDP algorithms is to optimize a value function, not to optimize time. The time taken is less than policy iteration since we do not need to solve a set of linear equations at every iteration in the way that policy iteration requires.

6 Video demo for running the code

My video demo for this project can be viewed on Youtube at https://youtu.be/JiFGHbxEPis or on Google Drive at https://drive.google.com/file/d/1fyw6xSbdpzbdAn-2AsNpzsd1XIaIBxt-/view?usp=sharing. The project contains a file called `readme.txt` that contains instructions on how the code can be run. The project needs to be run in a regular terminal for the input/output to be working correctly with the correct "-m memory_profiler" flag for the code to compile and record metrics properly. All necessary packages are installed in the virtual environment ("venv" folder) for Python 3.9.

7 References

[1]: Pyamaze (maze generator)
[2]: Bellman equation for policy iteration
[3]: Bellman equation for value iteration

8 Appendix

8.1 A* implementation

```
# Maze implementation library reference/credit:
Pyamaze (https://pypi.org/project/pyamaze/)

from queue import PriorityQueue
from search_algorithms.utility_functions
import set_neighbouring_square,
construct_path_from_dictionary


# A star algorithm implementation using
manhattan heuristic
def a_star_algorithm(maze):
    # Set the (x,y) coordinates of the starting
    square
    initial_maze_square = (maze.rows,
    maze.cols)
```

```
    # Before we start, lets keep track of
    explored squares and how we got there
    explored_squares = {}
    # Use a priority queue to keep track of
    nodes to explore
    nodes_to_explore = PriorityQueue()
    nodes_to_explore.put((manhattan_distance(
    initial_maze_square, (1, 1)),
    manhattan_distance(initial_maze_square, (1,
    1)), initial_maze_square))
    # G-score is the cost from the start
    g_score = dict.fromkeys(maze.grid,
    float('inf'))
    g_score[initial_maze_square] = 0
    # F-Score is the cost from the start +
    estimated cost (i.e. heuristic with
    manhattan distance)
    f_score = dict.fromkeys(maze.grid,
    float('inf'))
    f_score[initial_maze_square] =
    manhattan_distance(initial_maze_square, (1,
    1))
    # Start with initial maze square!
    maze_area_to_search = [initial_maze_square]
    # Keep looping until we have explored
    everything necessary
    while not nodes_to_explore.empty():
        # Get the 3rd element in tuple (i.e.
        square we are presently at)
        _, _, present_square =
        nodes_to_explore.get()
        # Add the current cell to the search
        space
        maze_area_to_search.append(
        present_square)
        # If we reached the target, break out
        of the loop!
        if present_square == (1, 1):
            break
        # Pyamaze takes single letter
        representations of each cardinal
        direction
        directions = {'North': 'N', 'East': 'E',
        'South': 'S', 'West': 'W'}
        # Otherwise, explore potentially
        neighbouring squares and update
        G-Score/F-Score if a better path is
        found
        for direction in directions.values():
            if maze.maze_map[present_square]
            [direction]:
                # Set the neighbouring square
                (x,y) based on the direction
                and the square we are
                presenting at
                neighbouring_square =
                set_neighbouring_square(
                direction, present_square)
                # Set a temporary G-Score and
                F-Score
                tentative_g_score =
                g_score[present_square]
                tentative_f_score =
                tentative_g_score +
                manhattan_distance(
                neighbouring_square, (1, 1))
                # If the tentative F-score
                (total estimated cost) for
                reaching 'neighbouring_square'
                is lower, update!
                if tentative_f_score <
                f_score[neighbouring_square]:
                    explored_squares[
                    neighbouring_square] =
                    present_square
                    f_score[neighbouring_square]
                     = tentative_f_score
                    g_score[neighbouring_square]
                     = tentative_g_score
                     nodes_to_explore.put((
                     tentative_f_score,
                     manhattan_distance(
                     neighbouring_square, (1,
                     1)), neighbouring_square))
    # Construct the path from the start to the
    target square
    path_to_target =
    construct_path_from_dictionary(
    initial_maze_square, explored_squares)
    return maze_area_to_search, path_to_target


# Manhattan distance between 2 points for the
heuristic (instead of Euclidean distance which
would underestimate it)
def manhattan_distance(starting_maze_square,
ending_maze_square):
    return abs(starting_maze_square[0] -
    ending_maze_square[0]) +
    abs(starting_maze_square[1] -
    ending_maze_square[1])
```

8.2 Depth-first search implementation

```
# Maze implementation library reference/credit:
Pyamaze (https://pypi.org/project/pyamaze/)

from search_algorithms.utility_functions
import set_neighbouring_square,
construct_path_from_dictionary


# Depth first search algorithm implementation
def dfs_algorithm(maze):
    # Set the (x,y) coordinates of the starting
    square
    initial_maze_square = (maze.rows,
    maze.cols)
    # Start with empty search space
    maze_area_to_search = []
    # Before we start, lets keep track of
    explored squares and how we got there
    explored_squares = {}
    # Start off with the initial square as the
    next square that is discovered
    discovered_squares = [initial_maze_square]
    subsequent_squares = [initial_maze_square]
    # Keep looping until we have explored
    everything necessary
    while len(subsequent_squares) > 0:
        # Get the next element in the list of
        squares to go to
        present_square = subsequent_squares[-1]
        # Remove the last element in the list
        of squares to go to (only keeping the
        1st to the 2nd last element)
        subsequent_squares =
        subsequent_squares[:-1]
        # If we reached the target, break out
        of the loop!
        if present_square == (1, 1):
            break
        # Pyamaze takes single letter
        representations of each cardinal
        direction
        directions = {'North': 'N', 'East': 'E',
        'South': 'S', 'West': 'W'}
        # Otherwise, explore potentially
        neighbouring squares
        for direction in directions.values():
            if maze.maze_map[present_square]
            [direction]:
                # Set the neighbouring square
                (x,y) based on the direction
                and the square we are
                presenting at
                neighbouring_square =
                set_neighbouring_square(
                direction, present_square)
                # If the neighbouring square has
                already been explored, ignore
                it
                if neighbouring_square in
                discovered_squares:
                    continue
                # Add the neighbour to the list
                of explored squares
                maze_area_to_search.append(
                present_square)
                # Add neighbour to the list of
                squares to explore next
                subsequent_squares.append(
                neighbouring_square)
                # Add neighbour to the list of
                discovered squares
                discovered_squares.append(
                neighbouring_square)
                # Update the DFS path between
                the (relationship between the
                neighbour and the present
                square)
                explored_squares[
                neighbouring_square] =
                present_square
    # Construct the path from the start to the
    target square
    path_to_target =
    construct_path_from_dictionary(
    initial_maze_square, explored_squares)
    return maze_area_to_search, path_to_target
```

8.3 Breadth-first search implementation

```
# Maze implementation library reference/credit:
Pyamaze (https://pypi.org/project/pyamaze/)

from search_algorithms.utility_functions
import set_neighbouring_square,
construct_path_from_dictionary


# Breadth first search algorithm implementation
def bfs_algorithm(maze):
```

```
    # Set the (x,y) coordinates of the starting
    square
    initial_maze_square = (maze.rows,
    maze.cols)
    # Start with empty search space
    maze_area_to_search = []
    # Before we start, lets keep track of
    explored squares and how we got there
    explored_squares = {}
    # Start off with the initial square as the
    next square that is discovered
    subsequent_squares = [initial_maze_square]
    discovered_squares = [initial_maze_square]
    # Keep looping until we have explored
    everything necessary
    while len(subsequent_squares) > 0:
        # Remove the first element off the list
        of the next squares we are going to
        present_square = subsequent_squares[0]
        del subsequent_squares[0]
        # If we reached the target, break out
        of the loop!
        if present_square == (1, 1):
            break
        # Pyamaze takes single letter
        representations of each cardinal
        direction
        directions = {'North': 'N', 'East': 'E',
        'South': 'S', 'West': 'W'}
        # Otherwise, explore potentially
        neighbouring squares
        for direction in directions.values():
            if maze.maze_map[present_square]
            [direction]:
                # Set the neighbouring square
                (x,y) based on the direction
                and the square we are
                presenting at
                neighbouring_square =
                set_neighbouring_square(
                direction, present_square)
                # If the neighbouring square has
                already been explored, ignore
                it
                if neighbouring_square in
                discovered_squares:
                    continue
                # Add the neighbour to the list
                of explored squares
                maze_area_to_search.append(
                neighbouring_square)
                # Add neighbour to the list of
                squares to explore next
                subsequent_squares.append(
                neighbouring_square)
                # Add neighbour to the list of
                discovered squares
                discovered_squares.append(
                neighbouring_square)
                # Update the BFS path between
                the (relationship between the
                neighbour and the present
                square)
                explored_squares[
                neighbouring_square] =
                present_square
    # Construct the path from the start to the
    target square
    path_to_target =
    construct_path_from_dictionary(
    initial_maze_square, explored_squares)
    return maze_area_to_search, path_to_target
```

8.4 Policy iteration implementation

```
# Maze implementation library reference/credit:
Pyamaze (https://pypi.org/project/pyamaze/)

from mdp_algorithms.utility_functions import
valid_actions, update_state_based_on_action,
set_initial_rewards


def policy_iteration_algorithm(maze):
    path = []
    current_state = (maze.rows, maze.cols)
    path.append(current_state)
    value_function = {}
    discount_factor = 0.9
    maze.rewards = {}

    # Set value function as 0 for all states
    for state in maze.maze_map:
        value_function[state] = 0

    # Set reward 1000 for the target state and
    -1 for non-target states
    set_initial_rewards(maze)

    # Complete policy iteration to find the
    optimal policy
    policy = policy_improvement(maze,
    value_function, discount_factor)
```

```
    # While the target is not reached yet, keep
    choosing the best action
    while current_state != (1, 1):
        action = max(policy[current_state],
        key=policy[current_state].get)
        current_state =
        update_state_based_on_action(
        current_state, action)
        path.append(current_state)
    return path


# Converge towards an optimal policy by
refining it
def policy_improvement(maze, value_function,
discount_factor):
    # Initialize a policy where each state has
    various actions with equal probability
    current_policy =
    initialize_policy_for_each_state(maze)
    # Find optimal policy
    while True:
        # Track if the policy changes during
        policy improvement
        policy_has_changed_during_improvement =
        True
        # Evaluate the current policy
        policy_evaluation(current_policy, maze,
        value_function, discount_factor,
        threshold=0.001)
        # Iterate over all states in the maze
        for state in maze.maze_map.keys():
            # Keep track of action values
            expected_values_of_all_actions = {}
            # Keep track of previous action
          action_chosen_by_policy_before_update
            = max(current_policy[state],
            key=current_policy[state].get)
            # Iterate through all valid actions
            for action in valid_actions(state,
            maze):
                next_state =
                update_state_based_on_action(
                state, action)
                # If this state is not valid in
                the maze, use the current state
                (agent stays in same position)
                if next_state not in
                maze.maze_map:
                    next_state = state
                # Calculate value of a action at
                 this next state
                expected_values_of_all_actions[
                action] =
                (maze.rewards.get(next_state,
                0) + discount_factor *
                value_function[next_state])
            # Calculate the best possible action
            action_with_highest_expected_value =
             max(expected_values_of_all_actions,
             key=expected_values_of_all_actions
             .get)
             # If the action is the best action,
             set its probability to 1, or 0
             otherwise
            for action in current_policy[state]:
                if action == action_with_highest
                 _expected_value:
                   current_policy[state][action]
                     = 1
                 else:
                   current_policy[state][action]
                     = 0
             # Update if the policy has changed
             to a better one
             if action_chosen_by_policy_before
             _update !=
             action_with_highest_expected_value:
                 policy_has_changed_during
                 _improvement = False
        # Break if the policy has improved
        if
        policy_has_changed_during_improvement:
            break
    return current_policy


# Initialize a policy where each state has
various actions with equal probability
def initialize_policy_for_each_state(maze):
    current_policy = {}
    for state in maze.maze_map.keys():
        current_policy[state] = {}
        # Start with uniform policy (set
        initial probabilities to be equal)
        for action in valid_actions(state,
        maze):
            # Set each action to have a
            probability of 1/total number of
            valid actions (to have equal
            probability)
            current_policy[state][action] = 1.0
            / len(valid_actions(state, maze))
```

```
    return current_policy


# Evaluate the value function until convergence
def policy_evaluation(policy, maze,
value_function, discount_factor, threshold):
    while True:
        max_change_in_value_function = 0
        for state in maze.maze_map.keys():
            estimated_state_value = 0
            # Iterate over all actions and
            their probabilities in the policy
            for action, probability_of_action
            in policy[state].items():
                next_state =
                update_state_based_on_action(
                state, action)
                # If this state is not valid in
                the maze, use the current state
                (agent stays in same position)
                if next_state not in
                maze.maze_map:
                    next_state = state
               # Use Bellman equation for state
                value estimation
                estimated_state_value =
                (estimated_state_value +
                probability_of_action *
                (maze.rewards.get(next_state,
                0) + discount_factor *
                value_function[next_state]))
            # Change in value function =
            absolute difference between new
            value - previous values for each
            state
            max_change_in_value_function =
            max(max_change_in_value_function,
            abs(estimated_state_value -
            value_function[state]))
            value_function[state] =
            estimated_state_value
        # Break if we reach convergence
        if max_change_in_value_function <
        threshold:
            break
```

8.5 Value iteration implementation

```
# Maze implementation library reference/credit:
Pyamaze (https://pypi.org/project/pyamaze/)

import random
from mdp_algorithms.utility_functions import
valid_actions, set_initial_rewards,
update_state_based_on_action


def value_iteration_algorithm(maze):
    path = []
    current_state = (maze.rows, maze.cols)
    path.append(current_state)
    potential_actions = {}
    decay = 0.9
    threshold = 0.005
    maze.rewards = {}

    # Set reward 1000 for the target state and
    -1 for non-target states
    set_initial_rewards(maze)

    # Iterate through all rows
    for row in range(1, maze.rows + 1):
        # Iterate through all columns
        for column in range(1, maze.cols + 1):
            present_square = (row, column)
            # Initialize no actions for the
            current square
            potential_actions[present_square] =
            []
            directions = {'North': 'N', 'East':
            'E', 'South': 'S', 'West': 'W'}
            # Go to all directions
            for direction in
            directions.values():
                # If the present square has a
                valid path in the current
                direction, save it as a
                potential action
               if maze.maze_map[present_square]
                [direction]:
                potential_actions[present_square]
                      = valid_actions((row,
                      column), maze)

    # Set value function with 10000 for the
    target and -1 for non-target squares
    value_function = {}
    for square in maze.maze_map:
        if square == (1, 1):
            value_function[square] = 10000
       elif square in potential_actions.keys():
            value_function[square] = -1

    # Set initial policy by setting random
    policies for each cell
```

```
    policy = {}
    for potential_action in
    potential_actions.keys():
        policy[potential_action] =
        random.choice(potential_actions[
        potential_action])

    while True:
        max_change_in_value_function = 0
        # Iterate through all squares in the
        maze
        for state in maze.maze_map:
            updated_value = 0
            previous_value =
            value_function[state]
            # Iterate through all potential
            actions for the present square
            for action in
            potential_actions[state]:
                next_state =
                update_state_based_on_action(
                state, action)
                value = (maze.rewards[state] +
                (decay * value_function[tuple(
                next_state)]))
                # Check if the action is better
                if value > updated_value:
                    # If so, store this action
                    and update the value
                    policy[state] = action
                    updated_value = value

            # Update the value of the current
            state
            value_function[state] =
            updated_value
            # Change in value function =
            absolute difference between new
            value - previous values for each
            state
            max_change_in_value_function =
            max(max_change_in_value_function,
            abs(previous_value -
            value_function[state]))
        # Break if we reach convergence
        if max_change_in_value_function <
        threshold:
            break

    # While the target is not reached yet, keep
    choosing the best action
    while current_state != (1, 1):
        action = policy[current_state]
        current_state =
        update_state_based_on_action(
        current_state, action)
        path.append(current_state)
    return path
```

8.6 Search algorithms utility functions

```
# Maze implementation library reference/credit:
Pyamaze (https://pypi.org/project/pyamaze/)

# Set the neighbouring squares depending on the
cardinal directions provided
def set_neighbouring_square(direction,
present_square):
    neighbouring_square = None
    if direction == 'N':
        neighbouring_square = (present_square[0]
        - 1, present_square[1])
    elif direction == 'E':
        neighbouring_square =
        (present_square[0], present_square[1] +
        1)
    elif direction == 'S':
        neighbouring_square = (present_square[0]
        + 1, present_square[1])
    elif direction == 'W':
        neighbouring_square =
        (present_square[0], present_square[1] -
        1)
    return neighbouring_square


# Construct the path from the dictionary of
explored squares
def construct_path_from_dictionary(
initial_maze_square, explored_squares):
    square = (1, 1)
    path_to_target = {}
    while square != initial_maze_square:
        path_to_target[explored_squares[square]]
        = square
        square = explored_squares[square]
    return path_to_target
```

8.7 MDP algorithms utility functions

```
# Maze implementation library reference/credit:
Pyamaze (https://pypi.org/project/pyamaze/)
```

```
# Get valid actions from current coordinates
def valid_actions(coordinates, maze):
    actions = maze.maze_map[(coordinates[0],
    coordinates[1])]
    valid_actions_to_take = []
    for direction, valid in actions.items():
        if valid:
            valid_actions_to_take.append(
            direction)
    return valid_actions_to_take


# Update a state/square based on an action
def update_state_based_on_action(state,
action):
    if action == 'N':
       updated_state = (state[0] - 1, state[1])
    elif action == 'E':
       updated_state = (state[0], state[1] + 1)
    elif action == 'S':
       updated_state = (state[0] + 1, state[1])
    elif action == 'W':
       updated_state = (state[0], state[1] - 1)
    else:
        # Unknown action supplied, so default
        to original state
        updated_state = state
    return updated_state


# Set initial rewards where the target square
has a reward of 1000 and non target squares
have -1
def set_initial_rewards(maze):
    for state in maze.maze_map:
        if state == (1, 1):
            maze.rewards[state] = 1000
        else:
            maze.rewards[state] = -1
```