Swinburne University of Technology
School of Science, Computing and Engineering Technologies

COS30019
Introduction To Artificial Intelligence

Assignment 1
Tree Based Search

Author: Axel Matthew Winjoto
Student ID: 103834503

Contents

Instructions:

How to run the program:

1. **CLI Mode**
    - Navigate to the directory that contains the “search.py” script.
    - Using the terminal, we can then execute the script which require 2 arguments.
    - The first argument would be the name of the text file we are using for the script.
    - The second argument would be to specify the search method we want to use such as BFS or DFS
    - An example of using BFS would be: “python search.py RobotNav-test.txt bfs”.
    - In which the output shown would be:
        - <Node (7, 0)> 34
        - ['down', 'right', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']
    - In the case where there would be no goal nodes found the output would be:
        - No goal is reachable; 11
    - The various search methods I’ve implemented, and the keywords needed to call them could be found below:
        - bfs (Breadth-First Search)
        - dfs (Deth-First Search)
        - gbfs (Greedy Best-First Search
        - astar (A* Search)
        - bidirectional (Bidirectional Search)
        - beam (Beam Search)

Introduction:

The Robot Navigation problem is one of various examples that introduces the challenge of pathfinding, where the end goal is to help an agent navigate from their starting point to a goal or destination while also avoiding obstacles and finding the most optimal and shortest path.

Terminologies Glossary:

- **Nodes**: Individual parts of a graph that can represent variables such as coordinates, state, positions or possible moves.
- **Graph**: Represents a group of nodes and the connection between them.
- **Tree Graph**: Graph formed by branching from a single root node, which in turn produces a hierarchical structure where every node would have one parent and either none or a collection of children.
- **Root Node**: The first node on a tree graph which would usually represent its starting point.
- **Path**: Represents the route the agent might be taking from the connection of one node to another.
- **Informed Search:** Search algorithms that contains additional information or heuristics that would help guide them towards the goal node more efficiently.
- **Uninformed Search:** Search algorithms that contains no additional information and could only distinguish a goal node from a non-goal node.
- **Breadth-First Search (BFS**): An uninformed search algorithm that explores all nodes at the current depth before attempting to move to the nodes at the next depth level.
- **Depth-First Search (DFS)**: An uninformed search algorithm that explores a node and their branch as deep as possible before going back and attempting the next nodes.
- **Greedy Best-First Search (GBFS)**: An informed search algorithm that explores the most direct path to the goal node.
- **A-Star Search (A* Search)**: An informed search algorithms that uses the total cost between the cost to reach the next node and the cost to the goal node to find the path with the least cost.
- **Bidirectional Search**: An uninformed search algorithm that runs two searches at the same time one from the agent and one from the goal node.
- **Beam Search**: An informed search algorithm that explores the graph by expanding the most promising nodes in the limited set.

Search Algorithms:

Breadth First Search (BFS)

- **Thoroughness**: BFS would always guarantees that the goal node would always be discovered.
- **Efficiency**: If all actions have the same costs BFS would always find the shortest path to the goal node especially when all cost between the nodes are similar.
- **Time Complexity**: BFS would always explore large number of nodes and the time complexity would depend on the depth of the solution where the deeper the solution is located the longer the BFS have to run.
- **Space Complexity**: Within the current depth BFS would store every single node explored which would consume significant space and memory.
- **RobotNav-test.txt case:** In the case of the robot navigation text file the BFS has an average run time of 0.002393 seconds with a total of 34 nodes explored.

Depth First Search (DFS)

- **Thoroughness**: DFS may be subject to infinite loops if the graph has a branch that goes on forever since DFS would always focus on the depth of one branch.
- **Efficiency**: DFS does not guarantee an optimal solution as its properties of focusing on the depth of one branch before moving on to another node would cause to find a long path instead of a shorter one.
- **Time Complexity**: If the depth of the graph and solution is short DFS could provide a faster time complexity than BFS.
- **Space Complexity**: The only advantage DFS has over BFS is that in terms of space complexity DFS is much more efficient as it only stores a single route from the root node to the end node/leaf node with the unexplored neighbouring nodes.
- **RobotNav-test.txt case:** In the case of the robot navigation text file the DFS has an average run time of 0.002000 seconds with a total of 26 nodes explored.

Greedy Best-First Search (GBFS)

- **Performance:** The performance of GBFS would be depends on the heuristic quality used.
- **Time complexity:** Since the performance is highly dependent on the different kind of heuristics used, it is possible for GBFS to be faster than BFS and DFS but only if the heuristic used optimal.
- **Efficiency:** It's not guaranteed to find the shortest as the heuristics used could be misleading.
- **RobotNav-test.txt case:** In the case of the robot navigation text file the GBFS has an average run time of 0.002200 seconds with a total of 23 nodes explored since there are obstacles involved that the heuristics wouldn't account for this would contribute in negatively impacting the performance and efficiency of GBFS.

A-Star Search (A* Search)

- **Optimality and Performance:** A* would always find the shortest path that also costs the least in terms of its heuristics.
- **Time Complexity:** If the goal node in the graph is located deep inside the graph, then A* search would be more efficient and faster than BFS.
- **Space Complexity:** A* would often consume a lot of memory.

- **RobotNav-test.txt case:** In the case of the robot navigation text file the A* search algorithm has an average run time of 0.002700 seconds with a total of 28 nodes explored.

Bidirectional Search (BFS)

- **Thoroughness:** Guaranteed to find the goal node only if both goal node and root node uses BFS
- **Time Complexity:** Faster than BFS since both node uses BFS in order to find each other which would lead them to meet in an intersecting node instead of the root node having to search for goal node.
- **Space Complexity:** Better than BFS as it involves both the agent and the goal node which means the root node only have to store nodes up to half than that of BFS.

- **RobotNav-test.txt case:** In the case of the robot navigation text file the Bidirectional search algorithm has an average run time of 0.001500 seconds with a total of 16 nodes explored.

Beam Search

- **Thoroughness:** Highly dependant on the amount of set its allowed to explore and since it might skip on exploring some part of the graph.
- **Time Complexity:** Could be very fast as it selects only a number of best nodes to explore on each depth level.
- **Space Complexity:** Consumes less memory as the nodes stored by beam search are only up to a limited number:
- **RobotNav-test.txt case:** In the case of the robot navigation text file the Bidirectional search algorithm has an average run time of 0.003000 seconds with a total of 28 nodes explored.

Implementations:

Grid Class Pseudocode:

```
class Grid
   Define the constructor with an optional filepath parameter
      If filepath is provided
         Call the load_grid_data method with the filepath
      Else
         Initialize an empty grid
         Set the agent_position to (0, 0)
         Initialize an empty set for goals

   Define the load_grid_data method with a filepath parameter
      Open the file specified by filepath for reading
      Read all lines from the file
      Extract grid dimensions from the first line, converting from string to integers
      Initialize the grid based on these dimensions, filling with empty strings
      Extract the agent's initial position from the second line, setting it in the grid
      Initialize the set for goals
      Extract goal positions from the third line, setting them in the grid and adding
to the goals set
      For each remaining line (assumed to describe obstacles)
         Extract obstacle properties (position and size)
         Fill the specified region in the grid with 'W' to denote walls

   Define the is_valid_move method with x, y coordinates
      Check if the coordinates are within the grid boundaries
      If within boundaries, check if the cell is neither a wall ('W') nor the agent's
starting point ('A')
      Return true if the move is valid, false otherwise
```

BFS Pseudocode:

```
class BFS
   Define a function search with parameters start and goals
      Initialize directions with tuples indicating movement and corresponding labels:
         (-1, 0) for "up"
         (0, -1) for "left"
         (1, 0) for "down"
         (0, 1) for "right"

      Create a queue initialized with the starting point and an empty path
      Initialize a visited set with the start point
      Create an empty list for explored nodes
      Initialize a counter for nodes explored

      While the queue is not empty
         Dequeue the current node and its path from the queue
         Increment nodes explored counter
         Add the current node to the explored nodes list with a 'explored' tag

         If the current node is one of the goals
            Return the current node, the path to reach it, all explored nodes, and the total nodes
explored

         Loop through each movement direction and label
            Calculate new coordinates by adding movement direction to current node coordinates

            If the move is valid on the grid and the new coordinates are not visited
               Add new coordinates to the visited set
               Add new coordinates to the explored nodes list with a 'visited' tag
               Enqueue the new coordinates and the updated path

      If no goal is found
         Return None, empty path, all explored nodes, and the total nodes explored
```

DFS Pseudocode:

```
class DFS(SearchAlgorithm)
   Define a function search with parameters start and goals
      Initialize movement directions for right, down, left, and up
      Create a stack initialized with the start position and an empty path list
      Initialize a visited set with the start position
      Initialize lists for explored nodes and a counter for nodes explored

      Increment nodes explored counter and add start position to explored nodes

      While the stack is not empty
         Pop the last node and path from the stack
         Add current node to explored nodes

         If current node is a goal
            Return the current node, path to it, explored nodes list, and count of nodes
explored

         Loop through each direction
            Calculate new coordinates based on the current node and direction
            If new coordinates are valid and not visited
               Mark new coordinates as visited
               Increment nodes explored counter
               Add new coordinates to explored nodes as 'visited'
               Push new coordinates and updated path onto the stack

      If no path to a goal is found
         Return None, empty path, explored nodes list, and nodes explored count
```

GBFS Pseudocode:

```
class GBFS(SearchAlgorithm)
   Define a function heuristic with parameter node
      Return the minimum Manhattan distance from the node to any goal

   Define a function search with parameters start and goals
      Initialize a priority queue
      Push the start node with its heuristic value and an empty path into the priority queue
      Initialize a visited set
      Initialize lists for explored nodes and a counter for nodes explored

      While the priority queue is not empty
         Pop the node with the lowest heuristic value, its current position, and path from the queue
         Add current node to explored nodes

         If current node is already visited
            Continue to next iteration

         Mark current node as visited

         If current node is a goal
            Return the current node, path to it, explored nodes list, and nodes explored count

         Loop through each possible move (up, left, down, right)
            Calculate new coordinates based on current position and move direction
            If the move is valid and the new coordinates are not visited
               Calculate heuristic for new coordinates
               Push new coordinates, updated path, and their heuristic into the queue
               Increment nodes explored counter
               Add new coordinates to explored nodes as 'visited'

      If no path to a goal is found
         Return None, empty path, explored nodes list, and nodes explored count
```

Search Algorithm Base Abstract Class

```
class SearchAlgorithm
   Define constructor with parameter grid
      Initialize grid attribute with the grid passed in

   Define an abstract function search with parameters start and goals
      This function is meant to be implemented by subclasses
```

A * Pseudocode:

```
class AStarSearch(SearchAlgorithm)
  Define a function heuristic with parameter node
    Return the minimum Manhattan distance from the node to any goal

  Define a function search with parameters start and goals
    Initialize a priority queue
    Compute initial heuristic value for start node
    Push tuple of initial heuristic value, initial cost (0), start node, and empty path into
the priority queue
    Initialize a visited set
    Initialize lists for explored nodes and a counter for nodes explored

    While the priority queue is not empty
      Pop the node with the lowest cost (f), along with its current cost (g), position,
and path from the queue
      Increment nodes explored counter
      Add current node to explored nodes

      If current node is already visited
        Continue to next iteration

      Mark current node as visited

      If current node is a goal
        Return the current node, path to it, explored nodes list, and nodes explored
count

      Loop through each possible move (up, left, down, right)
        Calculate new coordinates based on current position and move direction
        If the move is valid and new coordinates are not visited
          Calculate new cost (g) and heuristic (h) for new coordinates
          Compute new total cost (f) as sum of new g and h
          Push new coordinates, new g, updated path, and new f into the queue
          Add new coordinates to explored nodes as 'visited'

    If no path to a goal is found
      Return None, empty path, explored nodes list, and nodes explored count
```

Beam Search Pseudocode:

```
class BeamSearch(SearchAlgorithm)
  Define constructor with parameters grid and optional beam_width
(default to 3)
    Call the superclass constructor with grid
    Set beam_width

  Define heuristic function with parameter node
    Calculate and return the minimum Manhattan distance from node to
any goal

  Define search function with parameters start and goals
    If goals are empty
      Return None, empty list for paths, explored nodes, and node count

    Initialize a priority queue with the start node, its heuristic, and an
empty path
    Initialize a visited set
    Initialize lists for explored nodes and a counter for nodes explored

    While priority queue is not empty
      Sort priority queue by heuristic values
      Limit the number of nodes to process to the beam width
      Reset priority queue for the next level

      For each node in the current level
        Increment nodes explored counter
        Add current node to explored nodes as 'explored'
        Mark current as visited

        If current node is a goal
          Return current node, path to it, explored nodes, and node
count

        For each possible move from current node (up, left, down, right)
          Calculate new coordinates
          If move is valid and new coordinates not visited
            Compute path for new coordinates
            Append new coordinates, their heuristic, and new path to
priority queue
            Add new coordinates to explored nodes as 'visited'

    If no path to a goal is found
      Return None, empty path, explored nodes, and node count
```

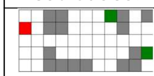
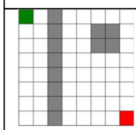
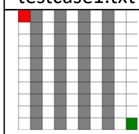
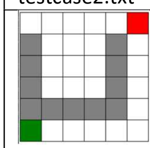
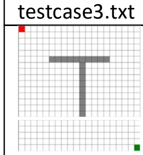
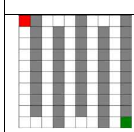
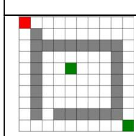
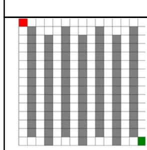
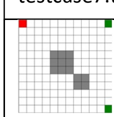
Bidirectional Pseudocode:

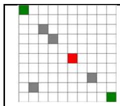
```
class BiDirectionalSearch(SearchAlgorithm)
  Define a function search with parameters start and goals
    If goals are empty
      Return None, empty list for moves, explored nodes, and node count

    Initialize two queues: front_queue starting from 'start' and back_queue starting from
each 'goal'
    Initialize two dictionaries: front_visited and back_visited to track paths from start and
goals respectively
    Initialize lists for explored nodes and a counter for nodes explored

    While both front_queue and back_queue are not empty
      Increment nodes explored counter

      If front_queue is not empty
        Dequeue from front_queue to get front_current and front_moves
        Add front_current to explored nodes as 'explored'

        If front_current is found in back_visited
          Retrieve goal_node and combine moves from both searches
          Return goal_node, combined_moves, explored nodes, and node count

        Explore neighboring nodes from front_current
          For each neighbor not in front_visited and is valid
            Add to front_visited and enqueue in front_queue
            Mark neighbor as 'visited' in explored_nodes

      If back_queue is not empty
        Dequeue from back_queue to get back_current and back_moves
        Add back_current to explored nodes as 'explored'

        If back_current is found in front_visited
          Retrieve goal_node and combine moves from both searches
          Return goal_node, combined_moves, explored nodes, and node count

        Explore neighboring nodes from back_current
          For each neighbor not in back_visited and is valid
            Add to back_visited and enqueue in back_queue
            Mark neighbor as 'visited' in explored_nodes

    If no intersection of paths is found
      Return None, empty list for moves, explored nodes, and node count
```

GridGUI Class Pseudocode:

```
class GridGUI
  Define constructor with optional grid parameter
    Initialize main window, frame, and canvas with scrollbars
    Initialize a button for opening files
    If a grid is provided, set it up; otherwise, initialize empty grid data

  Define create_buttons method
    Create and pack buttons for each search algorithm and file opening, linked
to their respective functions

  Define open_file method
    Open file dialog, create Grid object if file is selected, and set up grid

  Define setup_grid method
    Clear canvas, draw grid based on current grid data, create buttons, adjust
scrollable region

  Define draw_grid method with grid parameter
    Draw each cell on the canvas with color based on contents (agent, wall,
goal, empty)

  Define reset_and_search method with search_algorithm parameter
    Disable buttons, redraw grid, perform search, color nodes, show path or
display message if no goal found

  Define disable_all_buttons and enable_all_buttons methods
    Change state of all buttons to disabled or normal

  Define color_nodes_step_by_step method
    Sequentially color nodes, then perform callback function

  Define show_path method
    Visually trace path on the grid, updating colors of squares along the path

  Define run method
```

Testing:

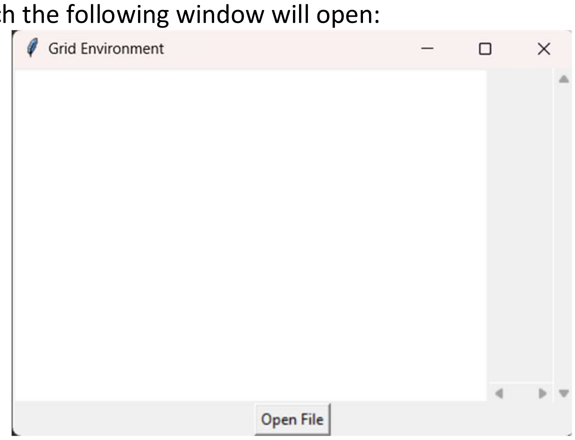
Test Cases:	BFS	DFS	GBFS	A*	Bidirectional	Beam
 RobotNav-text.txt	<Node (7, 0)> 34 ['down', 'right', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']	<Node (7, 0)> 26 ['down', 'right', 'right', 'right', 'down', 'right', 'right', 'up', 'up', 'up', 'right', 'right']	<Node (7, 0)> 23 ['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']	<Node (7, 0)> 28 ['down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up']	<Node (7, 0)> 16 ['down', 'right', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']	<Node (7, 0)> 28 ['right', 'down', 'right', 'right', 'right', 'up', 'up', 'right', 'right', 'right']
 testcase1.txt	No goal is reachable; 37	No goal is reachable; 36	No goal is reachable; 56	No goal is reachable; 57	No goal is reachable; 16	No goal is reachable; 61
 testcase2.txt	No goal is reachable; 11	No goal is reachable; 10	No goal is reachable; 10	No goal is reachable; 11	No goal is reachable; 10	No goal is reachable; 10
 testcase3.txt	<Node (0, 5)> 26 ['down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left']	<Node (0, 5)> 25 ['down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left']	<Node (0, 5)> 31 ['down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left']	<Node (0, 5)> 32 ['down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left']	<Node (0, 5)> 8 ['down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left']	No goal is reachable; 28
 Testcase4.txt	<Node (19, 19)> 382 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', …]	<Node (19, 19)> 362 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', 'up….]	<Node (19, 19)> 76 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down'…]	<Node (19, 19)> 631 ['down', 'right', 'right', 'right'…]	<Node (19, 19)> 181 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', ….]	<Node (19, 19)> 112 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'right'….]
 testcase5.txt	<Node (9, 9)> 28 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up'….]	<Node (9, 9)> 56 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', ….]	<Node (9, 9)> 55 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', ….]	<Node (9, 9)> 56 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', …	<Node (9, 9)> 28 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', ….	<Node (9, 9)> 55 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up'….]
 testcase6.txt	<Node (4, 4)> 61 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'right', 'right']	<Node (4, 4)> 28 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'right', 'right', 'down']	<Node (9, 9)> 41 ['right', 'right', 'right', 'right', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down']	<Node (4, 4)> 68 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'right', 'right', 'up', 'up', 'up']	<Node (4, 4)> 34 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up', 'up', 'up', 'up', 'right', 'right']	<Node (9, 9)> 45 ['right', 'right', 'down', 'right', 'right', 'up', 'right', 'right', 'right', 'down', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down']
 testcase7.txt	<Node (14, 14)> 135 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'right….]	<Node (14, 14)> 120 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down'….]	<Node (14, 14)> 39 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', …]	<Node (14, 14)> 144 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'right', ….]	<Node (14, 14)> 37 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down', 'down'….]	<Node (14, 14)> 122 ['right', 'right', 'down', 'right', 'right', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', ….]
 testcase8.txt	<Node (11, 0)> 71 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	<Node (11, 0)> 95 ['down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'down', 'right', 'right', 'up', 'up',…]	<Node (11, 0)> 23 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	<Node (11, 0)> 13 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	<Node (11, 11)> 28 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']	<Node (11, 0)> 31 ['right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right']

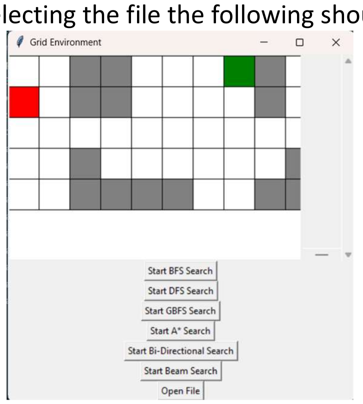
 Testcase9.txt	<Node (9, 9)> 92 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'right']	<Node (0, 0)> 26 ['up', 'up', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'left']	<Node (9, 9)> 22 ['right', 'right', 'right', 'right', 'down', 'down', 'down', 'down']	<Node (9, 9)> 37 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'right']	<Node (5, 5)> 23 ['down', 'down', 'down', 'right', 'right', 'right', 'right', 'down']	<Node (9, 9)> 23 ['down', 'down', 'down', 'down', 'right', 'right', 'right', 'right']

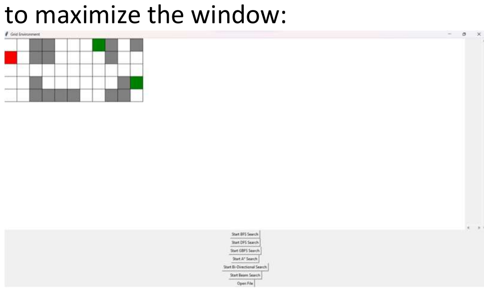
(Note: Some of the output produced are too long to put in the table so I've cut some of them out)

Features/Bugs/Missing and GUI research:

1. **GUI Mode:**

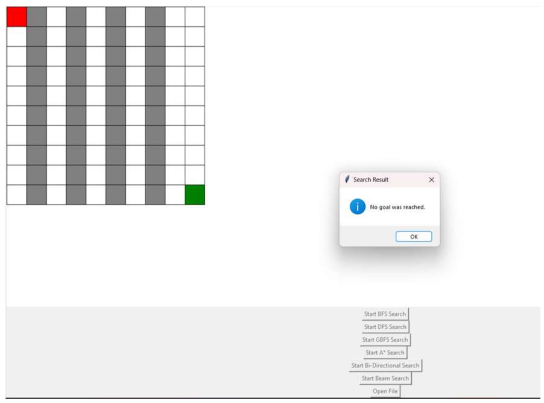
- Navigate to the directory that contains the "search.py" script.
- Using the terminal, we can then execute the script which for GUI require only 1 argument.
- The argument would just be a simple "gui" text.
- An example of activating the GUI mode would be: "python search.py gui".
- In which the following window will open:
    - 

- In order to show the grid, we would first need to click the open file button
- After clicking the open file, it will open file explorer where we would select the txt file we want to open and show in the GUI in this it would be the RobotNav-test.txt, where after selecting the file the following should be displayed:
    - 

- Currently its in minimized window format in order to have a better look we would have to maximize the window:
- 

- We can press the various search algorithm buttons which will show and display how each search algorithms would navigate the grid.
- In order to change file just click the open file button again and select the txt file we want to display to the GUI again.

2. Notes and Particulars of Implementation

- In GUI mode, the buttons to select the search algorithm will only appear after a text file is selected from file explorer.
- In order to be responsive, the GUI is configured with a scroll bar which would only work when a larger grid is displayed.
- Once a search method is started all buttons would be disabled in order to prevent overlapping of drawing and grids.
- If there are no goal nodes found, a message pop up will be shown indicating that no goal nodes are found:
    - 

- For the features missing is that currently I did not implement any method for the user to create their own custom maze and diagram, other than that so far I would like to think that I have fully implemented all features possible for the assignment and GUI extension.

Conclusion:

Overall for the navigation problem that was given to us with the text file RobotNav-test.txt I would initially say that A* would be the most optimal search method but after testing the runtime for each method and double checking the number of nodes explored I would say that for this scenario either Bidirectional or DFS are the best algorithm for this specific solution as the goal nodes are not located deep inside the graph tree and could be easily found using DFS and Bidirectional search. However, for general grid pathfinding I would still recommend the A* search method as they are the most optimal search method so far. In terms of performance improvement, I would try to find other methods of implementing higher quality heuristics to the A* search algorithm in order for it to be even more optimal in pathfinding the goal node.

Acknowledgement/Resources and References:

How to Implement BFS, Educative.IO:
https://www.educative.io/answers/how-to-implement-a-breadth-first-search-in-python
Helped me in understanding the overall method and overall guideline on how to add BFS class.

Bi-Directional Search Algorithms, GeeksforGeeks:
https://www.educative.io/answers/how-to-implement-a-breadth-first-search-in-python
Helped me in understanding the overall method and overall guideline on how to add Bi-directional class and helped me in learning of a new uninformed method.

A* Search Algorithms, GeeksforGeeks:
https://www.geeksforgeeks.org/a-search-algorithm/
Helped me in understanding the overall method and overall guideline on how to add and implement A*class to my code, also helped in understanding on how the heuristics of A* works.

GBFS Search Algorithms, GeeksforGeeks:
https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/
Helped in my implementation of the greedy best first class to the code.

Beam Search Algorithms, GeeksforGeeks:
https://www.geeksforgeeks.org/introduction-to-beam-search-algorithm/
Assisted me in the implementation of the beam search algorithm class.

Pathfinding Visualizer, Github, Clementmihailescu:
https://clementmihailescu.github.io/Pathfinding-Visualizer/
My GUI is based on how this pathfinding visualiser worked.

Pathfinding Visualizer, Vercel:
https://pathfinding-visualizer-nu.vercel.app
Another reference that I based my GUI on.