

Swinburne University Of Technology

Semester 1, 2023

COS30019

Introduction to Artificial Intelligence



Assignment Report

Robot Navigation Problem

Student Name: Nghia Hieu Pham

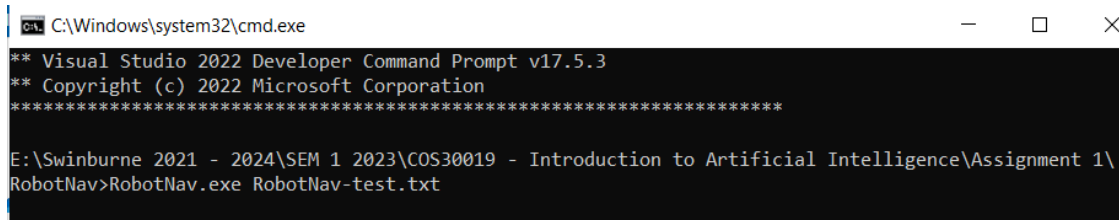
Student ID: 103533868

Table of contents (TOC)

Table of contents (TOC)	2
Instructions	3
Introduction	5
Search Algorithms	6
Uninformed Search	6
Breadth-First Search (BFS)	6
Depth-First Search (DFS)	6
Bidirectional Search	6
Informed Search	6
Greedy Best-First Search (GBFS)	7
A* Search	7
Iterative Deepening A* Search (IDA*)	7
Implementation	8
Uninformed Search	8
Breadth-First Search (BFS)	8
Depth-First Search (DFS)	8
Bidirectional Search	9
Informed Search	10
Greedy Best-First Search (GBFS)	10
A* Search	11
Iterative Deepening A* Search (IDA*)	11
Features/Bugs/Missing	13
Research.....	14
Graphical User Interface (GUI).....	14
Conclusion.....	15
Acknowledgements/Resources.....	16
References	16

Instructions

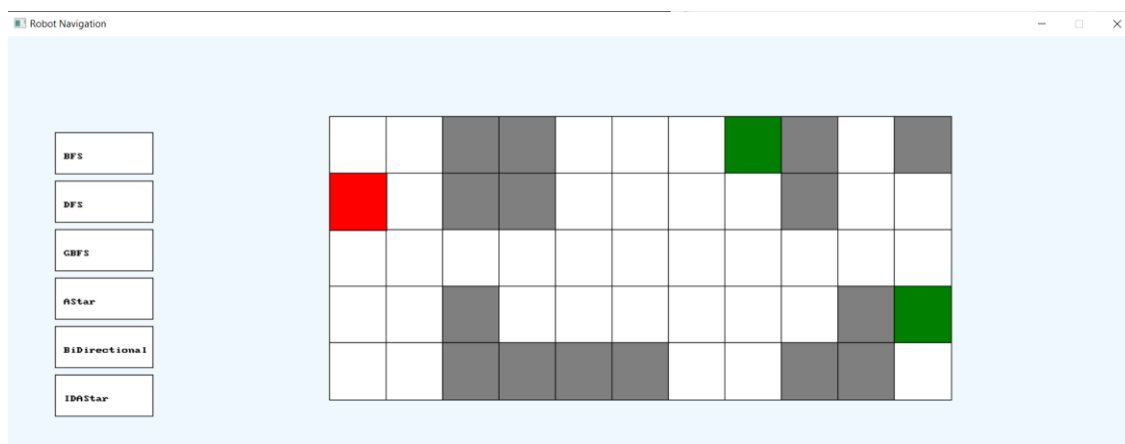
My program operates from a DOS command-line interface that supports batch testing. To run the program, type “*RobotNav.exe <filename>*”, where <filename> is the name of the test file, into the command prompt as shown in the image below:



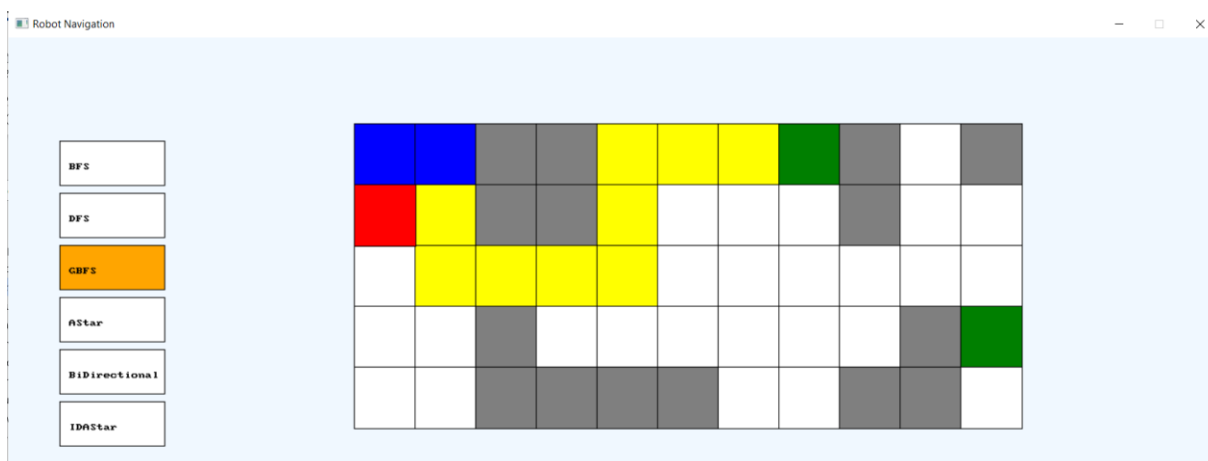
```

C:\Windows\system32\cmd.exe
** Visual Studio 2022 Developer Command Prompt v17.5.3
** Copyright (c) 2022 Microsoft Corporation
*****
E:\Swinburne 2021 - 2024\SEM 1 2023\COS30019 - Introduction to Artificial Intelligence\Assignment 1\
RobotNav>RobotNav.exe RobotNav-test.txt
  
```

After that, a window will show up with buttons used for selecting search algorithm on the left and the map showing robot's initial position (red), walls (gray), goals (green) on the right.



To find path to the solution, simply left click on a desired search algorithm button (the clicked button will turn orange). The path from robot's initial position to a goal is colored yellow and visited cells (but not solution path) is colored blue.



In addition, an output is printed in the console with the following format:

```
"<filename> <method> <number_of_nodes>  
<path>"
```

<filename>: Name of the test file

<method>: Name of the search algorithm used

<number_of_nodes>: Total number of searched nodes

<path>: Solution path from initial position to goal in 4 directions (up, left, down, right) or "No solution found" when no solution is found

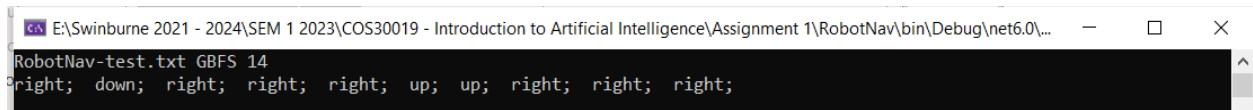
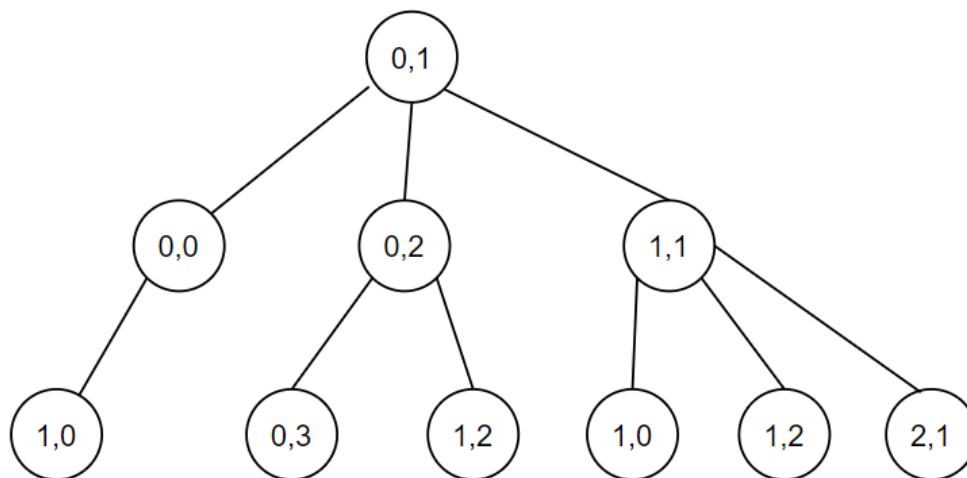


Figure 1: Example Console Output

Introduction

The Robot Navigation Problem is a common maze problem that tests different search algorithms on search trees. The environment is an $N \times M$ grid (where $N > 1$ and $M > 1$) with a number of walls blocking some cells. The robot is initially located in one of the empty cells and required to find a path to visit one of the designated cells of the grid called “goals”. The robot can either move left, down, up or right, but **when all else is equal**, the robot must try to move up before left, before down, before right. Furthermore, **when all else is equal**, two nodes H_1 and H_2 on two different branches of the search tree should be expanded according to the chronological order.

There are 3 uninformed and 3 informed search algorithms implemented in the program, which are Breadth-First Search (BFS), Depth-First Search (DFS), Bidirectional Search, Greedy Best-First Search (GBFS), A* Search, and Iterative Deepening A* Search (IDA*). All the implemented algorithms expand nodes in a tree, which is “a special type of graph that is connected and acyclic, meaning that there are no cycles in the graph. In a tree, there is a unique path between any two vertices, and there is a single vertex called the root that is used as the starting point for traversing the tree.”⁽¹⁾ The concept of a tree is represented by following figure taking [2, 2] as the initial position of the robot:



Furthermore, the program is written in object-oriented programming style (in C#) that sees everything as an object that knows things and can do things.

Search Algorithms

Uninformed Search

“Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.”⁽²⁾

Breadth-First Search (BFS)

BFS algorithm starts searching from the root node of the tree and expands all successor node at the current depth before moving to nodes of next depth.

Advantages	Disadvantages
<ul style="list-style-type: none"> - Provides a solution (if exists). - Provides the optimal solution which requires the least number of steps (if there are more than one solution). 	<ul style="list-style-type: none"> - Requires lots of memory. - Needs lots of time if the goal node is far away from the root node.

Depth-First Search (DFS)

DFS starts searching from the root node and follows each path to its greatest depth node before moving to the next path.

Advantages	Disadvantages
<ul style="list-style-type: none"> - Requires very less memory. - In ideal condition, DFS can take less time to reach the goal node than BFS algorithm. 	<ul style="list-style-type: none"> - Not guaranteed to find a solution (may go into infinite loop). - If more than one solution exists, DFS is not guaranteed to find the optimal solution. - Needs lots of time if the goal node is far away from the root node.

Bidirectional Search

Bidirectional algorithm runs two simultaneous searches: forward search from root node toward goal node and backward search from goal node toward root node. The search terminates when two searches meet at the same node.

Advantages	Disadvantages
<ul style="list-style-type: none"> - It is optimal if BFS is used for search and paths have uniform cost. - It is complete if BFS is used in both searches. 	<ul style="list-style-type: none"> - It requires lots of memory to store nodes from 2 simultaneous searches

Informed Search

Informed search algorithm contains information such as how far we are from the goal, path cost, how to reach to goal node, etc. This information helps agents evaluate the quality of states to plan a better

exploration. “The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.”⁽³⁾

Greedy Best-First Search (GBFS)

GBFS algorithm always selects the path which appears best at that moment. In the GBFS algorithm, we expand the closest node to the goal first, which is estimated by heuristic function $f(n) = h(n)$, where $h(n)$ is the estimated cost from node n to the goal.

Advantages	Disadvantages
- GBFS algorithm is more efficient than BFS and DFS algorithms.	- It is not guaranteed to find the solution (may get stuck in a loop) - It only looks for “short-term” goal, so it is not guaranteed to find the optimal solution (if there are more than one)

A* Search

A* search is similar to GBFS. The only difference between A* search and GBFS is that A* search also evaluate the cost to reach the node n from the root node in its heuristic function:

$$f(n) = h(n) + g(n)$$

$h(n)$: estimated cost from node n to the goal

$g(n)$: cost from the root node to node n

Advantages	Disadvantages
- Completeness: It is guaranteed to find the solution. - Optimality: It is guaranteed to find the optimal solution (if there are more than one)	- A* search requires lots of memory, so it is not suitable for large-scale problems.

Iterative Deepening A* Search (IDA*)

Fundamentally, IDA* is DFS algorithm using A* Search’s heuristic function to limit the depth of the DFS algorithm. Because IDA* performs all operations that A* Search does but uses less memory, it is a memory-limited version of A* Search.

Advantages	Disadvantages
- IDA* is guaranteed to find the optimal solution if one exists. - IDA* uses less amount of memory compared to A* search. - IDA* avoids the exponential time complexity of DFS by gradually increasing the depth limit.	- IDA* does not keep track of the visited nodes so it explored the visited nodes multiple times. - Because IDA* explored the visited nodes again and again, it may be slower than other search algorithms.

Implementation

First of all, the environment is created using the data parsed from the given text file containing size of the map, robot's initial position, goals' positions, and walls' positions. The environment's map is a 2D array of Cell object, which know its type ("Path", "Goal", "Wall", "Solution") and knows whether it is visited or not by using Boolean variable `_visited`. Moreover, the environment has an essential overloaded method (`discoverMoveSet()`) which will return a list of new possible states of the robot given the current state.

An abstract class named `SearchAlgorithm` was created as a template to flexibly implement different search algorithms. Every search algorithm has `SolveProblem()` method that contains the searching process and output, which will be displayed when the searching process terminated.

Furthermore, the estimated cost from a node to the goal is calculated using the Manhattan distance formula, which is the best approach for this assignment, as the movements were limited to only horizontal and vertical.

Uninformed Search

Breadth-First Search (BFS)

The BFS's implementation is expressed by the following pseudo code:

```
function SolveProblem ()
    frontier ← INSERT(MAKE-NODE(INITIAL-STATE))
    currentnode = null;
    searched = 0;
    solution_found = true;
    while (!EMPTY?(frontier)) do
        currentnode ← REMOVE-FIRST(frontier)
        searched++;
        if (GOAL-TEST[currentnode]) then break;
        MARK-VISITED[currentnode]
        frontier ← INSERT- ALL(EXPAND(currentnode))
        if EMPTY?(frontier) then
            solution_found = false
            break;
    DISPLAYSOLUTION(currentnode, searched, solution_found)
```

Queue<> type is used for frontier to implement a First-In-First-Out (FIFO) list.

Depth-First Search (DFS)

The DFS's implementation is expressed by the following pseudo code:

```
function SolveProblem ()
    frontier ← INSERT(MAKE-NODE(INITIAL-STATE))
    currentnode = null;
```



```

searched = 0;
solution_found = true;
while (!EMPTY?(frontier)) do
    currentnode ← REMOVE-FIRST(frontier)
    searched++;
    if (GOAL-TEST[currentnode]) then break;
    MARK-VISITED[currentnode]
    frontier ← INSERT-ALL(EXPAND(currentnode))
    if EMPTY?(frontier) then
        solution_found = false
        break;
DISPLAYSOLUTION(currentnode, searched, solution_found)

```

The DFS searching process is similar to BFS. The difference is that DFS uses Stack<> type for frontier to implement a Last-In-First-Out (LIFO) list.

Bidirectional Search

The Bidirectional Search's implementation is expressed by the following pseudo code:

```

function SolveProblem ()
    r_frontier ← INSERT(MAKE-NODE(ROBOT-INITIAL-STATE))
    g_frontier ← INSERT(MAKE-NODE(FIND-NEAREST-GOAL))

    currentnode_r = null;
    currentnode_g = null;

    visited_r = bool[MAP.WIDTH, MAP.HEIGHT]
    visited_g = bool[MAP.WIDTH, MAP.HEIGHT]

    visitedrobot = List<node>
    visitedgoal = List<node>
    searched = 0;
    solution_found = false;
    while (!EMPTY?(r_frontier) && !EMPTY?(g_frontier)) do
        currentnode_r ← REMOVE-FIRST(r_frontier)
        visitedrobot ← INSERT(currentnode_r)
        searched++;
        currentnode_g ← REMOVE-FIRST(g_frontier)
        visitedgoal ← INSERT(currentnode_g)
        searched++;

        MARK-VISITED(visited_r)
        MARK-VISITED(visited_g)

        if GOAL-TEST(visited_r, visited_g) then
            solution_found = true

```

```

    if (solution_found) then
        RETRIVE-INTERSECTED-NODE(visitedrobot)
        RETRIVE-INTERSECTED-NODE(visitedgoal)
        break;
    r_frontier ← INSERT-ALL(EXPAND(currentnode_r))
    g_frontier ← INSERT-ALL(EXPAND(currentnode_g))
    if (EMPTY?(r_frontier) && EMPTY?(g_frontier)) then
        break;
    DISPLAYSOLUTION(currentnode_r, currentnode_g, searched, solution_found)

```

The Bidirectional Search is implemented by running two BFS Search simultaneously: one forward search from the robot's initial position toward the goal and one backward search from the goal toward the robot's initial position. The visited_r and visited_g arrays of Boolean are used to mark the visited nodes. The solution will be found once when two BFS searches intersect at a node, which has coordinate [j, i] that satisfies (visited_r[j, i] == true && visited_g[j, i] == true). Then the intersected nodes from the forward and backward search will respectively be retrieved by using the visitedrobot and visitedgoal arrays, which store visited nodes of each search. The use of visitedrobot and visitedgoal arrays is necessary because the forward and backward searches might not arrive at the intersected node at the same time.

Informed Search

Greedy Best-First Search (GBFS)

The Greedy Best-First Search implementation is expressed by the following pseudo code:

```

function SolveProblem ()
    goal = FIND-NEAREST-GOAL();
    frontier ← INSERT(MAKE-NODE(INITIAL-STATE), CALCULATE-COST(INITIAL-STATE, goal))
    currentnode = null;
    searched = 0;
    solution_found = true;
    while (!EMPTY?(frontier)) do
        currentnode ← DEQUEUE(frontier)
        searched++;
        if (GOAL-TEST[currentnode]) then break;
        MARK-VISITED[currentnode]
        frontier ← INSERT-ALL(EXPAND(currentnode, goal))
        if EMPTY?(frontier) then
            solution_found = false
            break;
    DISPLAYSOLUTION(currentnode, searched, solution_found)

```

The search algorithm finds the nearest goal's position to the robot's initial position by calling findNearsetGoal() method. That goal will then be used to calculate the cost of each node. Notably, the frontier has type of PriorityQueue<> which will automatically sort the nodes based on their costs in an

ascending order. Furthermore, when being dequeued, the PriorityQueue<> will return the highest priority node (lowest cost).

A* Search

The A* searching process is the similar to GBFS's searching process which also makes use of the PriorityQueue<> type. The only difference is that the cost from robot's initial position to node n's position is also considered. However, in this assignment, the cost for robot to move either up, left, down, or right is the same (and equals 1). Therefore, in this assignment, A* Search's nodes evaluation is the same with GBFS's nodes evaluation, which will produce the same solution.

Iterative Deepening A* Search (IDA*)

The IDA* implementation is expressed by the following pseudo code:

```
function SolveProblem ()
    goal = FIND-NEAREST-GOAL();
    rootnode = MAKE-NODE(INITIAL-STATE), CALCULATE-COST(INITIAL-STATE, goal);
    f_score = GET-COST(rootnode)
    _pruned = PriorityQueue<node, int>
    currentnode = null;
    searched = 0;
    solution_found = true;

    frontier ← INSERT(rootnode)
    while (!EMPTY?(frontier)) do
        currentnode ← REMOVE-FIRST(frontier)
        searched++;
        if (GOAL-TEST[currentnode]) then break;
        MARK-VISITED[currentnode]
        frontier ← INSERT-ALL(EXPAND(currentnode, goal, 1))
        if (EMPTY?(frontier)) then
            if (EMPTY?(_pruned)) then
                solutionfound = false;
                break;
            f_score = GET-COST(DEQUEUE(_pruned))
            frontier ← INSERT(rootnode)
            MARK-UNVISITED
    DISPLAYSOLUTION(currentnode, searched, solution_found)
```

First of all, the search algorithm also finds the nearest goal's position to the robot's initial position and use that goal to calculate the cost of each node based on A* Search's heuristic function. Then the searching process is the same with DFS's, but the depth is limited by f_score. In specific, any node that has cost higher than f_score will be added to the _pruned PriorityQueue<>. If the frontier is empty and the goal node is not yet found, the f_score will be updated to the node with lowest cost in the _pruned

PriorityQueue<> and the search starts all over again from the root node with the updated f_score value. There will be no solution if the frontier and `_pruned` are empty, and the goal node is not found.

Search Algorithms Comparing

The following image shows the outputs of 6 search algorithms printed to the console using 2 different test files.

```

C:\Windows\system32\cmd.exe - RobotNav.exe RobotNav-test.txt
*****
E:\Swinburne 2021 - 2024\SEM 1 2023\COS30019 - Introduction to Artificial Intelligence\Assignment 1\RobotNav>RobotNav.exe
RobotNav-test.txt
RobotNav-test.txt BFS 113
down; right; right; right; right; up; up; right; right; right;

RobotNav-test.txt DFS 28
up; right; down; down; right; right; down; right; up; up; up; right; down; down; down; right; up; up;
up; right;

RobotNav-test.txt GBFS 14
right; down; right; right; right; up; up; right; right; right;

RobotNav-test.txt A Star 14
right; down; right; right; right; up; up; right; right; right;

RobotNav-test.txt Bidirectional 54
down; right; right; right; right; up; up; right; right; right;

RobotNav-test.txt IDA Star 26
up; right; down; down; right; right; down; right; up; up; up; right; down; down; down; right; up; up;
up; right;

```

Fig 2.1: Outputs for first test file

```

C:\Windows\system32\cmd.exe - RobotNav.exe RobotNav-test2.txt
*****
** Visual Studio 2022 Developer Command Prompt v17.5.3
** Copyright (c) 2022 Microsoft Corporation
*****
E:\Swinburne 2021 - 2024\SEM 1 2023\COS30019 - Introduction to Artificial Intelligence\Assignment 1\RobotNav>RobotNav.exe
RobotNav-test2.txt
RobotNav-test2.txt BFS 203
up; right; right; right; right; right; right; right; right; right; up; up;

RobotNav-test2.txt DFS 47
up; up; up; right; down; down; down; left; down; right; right; right; right; right; right; up; up;
up; up; up; right; down; down; down; down; down; right; right; right; up;

RobotNav-test2.txt GBFS 17
down; right; down; right; right; right; right; right; right; right; right; right; up;

RobotNav-test2.txt A Star 17
down; right; down; right; right; right; right; right; right; right; right; right; up;

RobotNav-test2.txt Bidirectional 92
up; right; right; right; right; right; right; right; right; right; right; right; down; down;

RobotNav-test2.txt IDA Star 40
down; right; down; right; right; right; right; right; up; up; up; up; up; right; down; down; down; down;
down; right; right; right; up;

```

Fig 2.2: Outputs for second test file

According to the results, despite of having the highest number of searched nodes, the BFS algorithm always found the optimal solutions. The DFS and IDA Star algorithms had much less nodes, but they did not produce optimal solution (IDA Star was likely to produce better solution than DFS). The Bidirectional Search used less nodes than the BFS and it almost always produce the optimal solution, making it better than the BFS. Finally, the A Star and GBFS had the least number of nodes and almost always produce the optimal solution, making them the best search algorithms in this problem.

Features/Bugs/Missing

I am currently missing the visualizer to show the changes happening to the search tree in my Graphical User Interface (GUI). In my GUI class, I implemented a method called `DrawVisitedCell()` to color the a cell blue when it is visited. Then I tried to call that method in the `SolveProblem()` method to color the cell blue right after it is visited but it did not work out. Instead, I just managed to color the visited cell blue after the searching process is done.

The program may run into bugs if one of the robot's initial position, walls' position, goals' position is the same with one other.

There are some limitations in my program. Firstly, if there are multiple goals, the search algorithms that require a goal's position will be provided with the nearest goal position to the robot's initial position using Manhattan distance formula. This approach may cause those algorithms to produce not optimal solution in case when the nearest goal is unreachable (blocked by walls). In addition, my program marks nodes visited slightly different than the standard search algorithm. In specific, when a node is searched, the cell with the according coordinates with that node will be marked visited. Therefore, duplication of the same node in different branches of the search tree is not allowed in my program. However, in the context of this assignment, as the path cost for moving up, left, down, or right is equal, the solution produced by the implemented algorithms will not be affected and will be the same with solution produced with the standard node-visited marking.

Conclusion

In overall, A* Search is the best search algorithm for this type of problem, especially when there are different path costs. Even though other algorithms may solve problems faster in some map layouts, they are not always guaranteed to find the optimal solution. Regarding performance improvements, I can think of allowing the robot to move in diagonal movements, which could potentially lower the searched nodes and produce better solution. In addition, I believe that I can save more memory by avoiding loops with smarter classes and methods design. Moreover, I would like to try combining different uninformed and informed search algorithms to find the combination with the best performance.

Acknowledgements/Resources

JavaTPoint <<https://www.javatpoint.com/>>: This website helped me better understand the concepts as well as disadvantages and advantages of uninformed and informed searches.

GeeksforGeeks <<https://www.geeksforgeeks.org/>>: I learned how to implement the Bidirectional Search and Iterative Deepening A* on this website.

SplashKit <<https://splashkit.io/>>: The documentation for the SplashKit library, which I used to create the GUI.

Microsoft .NET documentation: The documentation helped me understand and make better use of Stack<>, Queue<>, PriorityQueue<> type.

References

(1) Romin, V 2023, *Difference between graph and tree*, GeeksforGeeks, viewed 16 April 2023, <<https://www.geeksforgeeks.org/difference-between-graph-and-tree/>>.

(2) JavaTPoint, *Uninformed Search Algorithms*, JavaTPoint, viewed 16 April 2023, <<https://www.javatpoint.com/ai-uninformed-search-algorithms/>>.

(3) JavaTPoint, *Informed Search Algorithms*, JavaTPoint, viewed 16 April <<https://www.javatpoint.com/ai-informed-search-algorithms/>>.