Introduction To Artificial Intelligence

-by Fardaad Khan(ID: 103801976)

Assignment – 1

Robot Navigation Problem

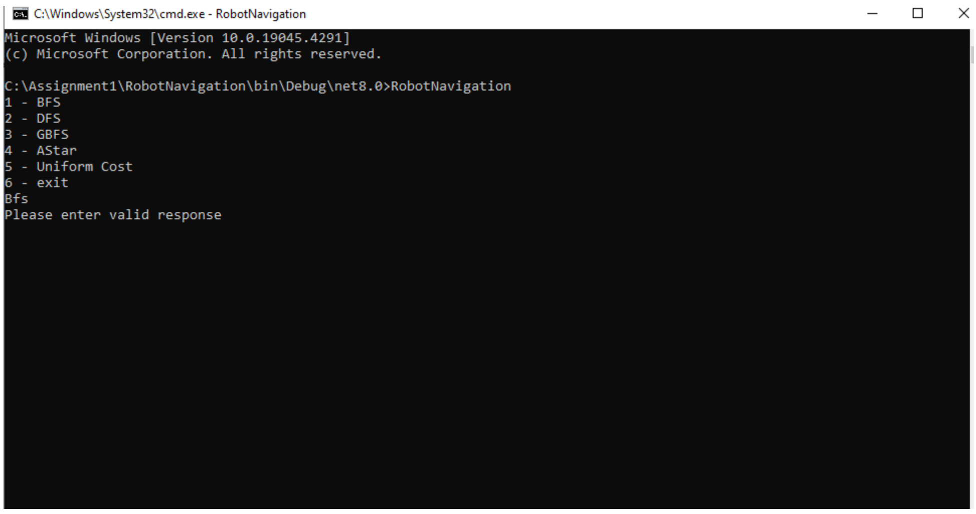
Table of Contents

Instructions

This program is developed in C#, it is a high-level programming language that supports a variety of paradigms. C# includes static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object oriented(class-based), and component-oriented programming disciplines.

The project can run in the terminal as follows:

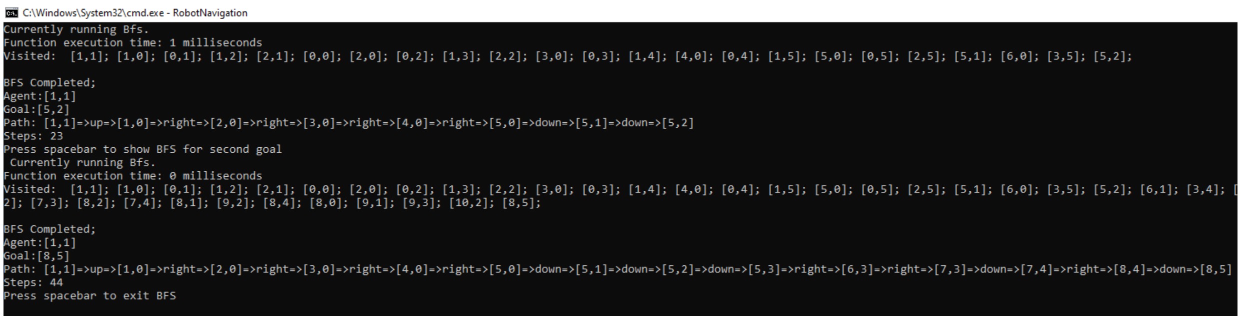
1) Open cmd (command prompt)
2) Navigate to C:\Assignment1\RobotNavigation\bin\Debug\net8.0>
3) Then type RobotNavigation and press enter.
4) A list of Search operations will be displayed.
5) **PLEASE NOTE:** The search letters are case sensitive so please type in 'bfs' instead of 'BFS' , 'Bfs' etc.



```
C:\Windows\System32\cmd.exe - RobotNavigation
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Assignment1\RobotNavigation\bin\Debug\net8.0>RobotNavigation
1 - BFS
2 - DFS
3 - GBFS
4 - AStar
5 - Uniform Cost
6 - exit
Bfs
Please enter valid response
```

Figure 1 Error shown if typed the wrong command.

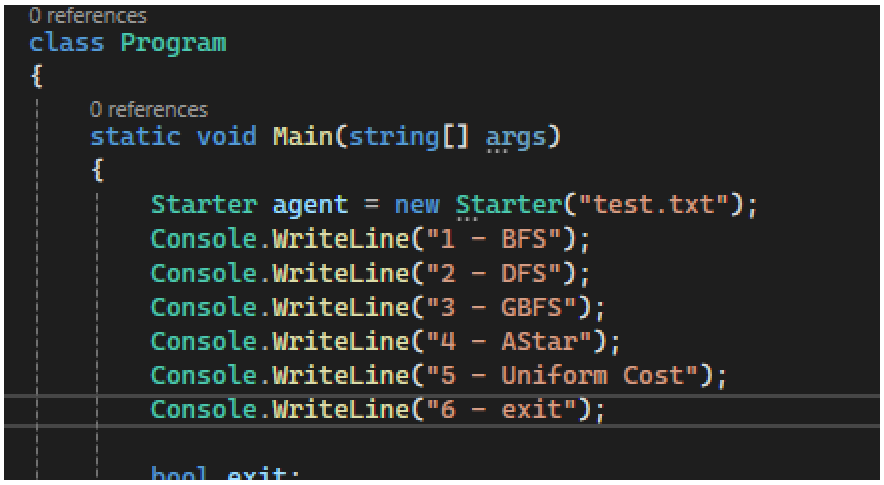
6) After typing 'bfs' in the terminal the program will generate the agent start position, goal state, path taken, steps taken and the search taken place, as shown below.



```
C:\Windows\System32\cmd.exe - RobotNavigation
Currently running Bfs.
Function execution time: 1 milliseconds
Visited:  [1,1]; [1,0]; [0,1]; [1,2]; [2,1]; [0,0]; [2,0]; [0,2]; [1,3]; [2,2]; [3,0]; [0,3]; [1,4]; [4,0]; [0,4]; [1,5]; [5,0]; [0,5]; [2,5]; [5,1]; [6,0]; [3,5]; [5,2];

BFS Completed;
Agent:[1,1]
Goal:[5,2]
Path: [1,1]=>up=>[1,0]=>right=>[2,0]=>right=>[3,0]=>right=>[4,0]=>right=>[5,0]=>down=>[5,1]=>down=>[5,2]
Steps: 23
Press spacebar to show BFS for second goal
 Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [1,1]; [1,0]; [0,1]; [1,2]; [2,1]; [0,0]; [2,0]; [0,2]; [1,3]; [2,2]; [3,0]; [0,3]; [1,4]; [4,0]; [0,4]; [1,5]; [5,0]; [0,5]; [2,5]; [5,1]; [6,0]; [3,5]; [5,2]; [6,1]; [3,4]; [
2]; [7,3]; [8,2]; [7,4]; [8,1]; [9,2]; [8,4]; [8,0]; [9,1]; [9,3]; [10,2]; [8,5];

BFS Completed;
Agent:[1,1]
Goal:[8,5]
Path: [1,1]=>up=>[1,0]=>right=>[2,0]=>right=>[3,0]=>right=>[4,0]=>right=>[5,0]=>down=>[5,1]=>down=>[5,2]=>down=>[5,3]=>right=>[6,3]=>right=>[7,3]=>down=>[7,4]=>right=>[8,4]=>down=>[8,5]
Steps: 44
Press spacebar to exit BFS
```

Figure 2 Screenshot of the BFS search

7) To run different test cases, we need to manually change the .txt file located in the following file: Assignment1/RobotNavigation/bin/Debug/net8.0/ Here the test.txt file is located and the new test cases can be inserted in here. Another way would be to manually change the code, but it will require a lot of time, it can be done using the program.cs file and changing the starter agent and adding the new file name as shown below:



```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Starter agent = new Starter("test.txt");
        Console.WriteLine("1 - BFS");
        Console.WriteLine("2 - DFS");
        Console.WriteLine("3 - GBFS");
        Console.WriteLine("4 - AStar");
        Console.WriteLine("5 - Uniform Cost");
        Console.WriteLine("6 - exit");

        bool exit;
```

Figure 3 Changing Test case

8) Since I have submitted the whole program file it will be easier for us to just navigate to the test.txt file and change the values in there and then run the code.

9) PLEASE NOTE : In order to run the search operations please use the following key words-

- **BFS search : Type 'bfs' in command prompt.**
- **DFS search: Type 'dfs' in command prompt.**
- **GBFS search: Type 'gbfs' in command prompt.**
- **A* Search: Type 'astar' in command prompt.**
- **Uniform search: Type 'uniform' in command prompt.**
- **Dijkstra's search : Type 'dsearch' in command prompt.**

Introduction

Intelligent autonomous robots can perform tasks and make decisions without human intervention. They are equipped with sensors, processors, and software, allowing them to Individuals perceive their surroundings, analyze data, and take actions to achieve predetermined objectives. They are essential for various applications such as space, transportation, industry, agriculture, healthcare, exploration, and the military. Mobile robots can perform dangerous, repetitive, or impossible tasks, such as material handling, disaster relief, patrolling, and rescue missions, more efficiently and accurately than humans. Sensor data must be processed and interpreted through computer vision and machine learning techniques. After perceiving its environment, the robot must decide on actions based on its goals and surroundings. It has gathered information. This requires advanced algorithms for planning, decision-making, and control, as well as adaptability to new situations.

An autonomous robot must be able to navigate both static and dynamic environments. Mobile robot navigation aims to move a robot efficiently and securely from its starting point to its destination in a cluttered environment. It prioritizes safety and shortest path length. Researchers investigated various techniques for robot navigation path planning in this area. Machines now perform physical tasks like car and building construction, reducing the need for human labor. Machines must think and make judgements like humans, in addition to performing physical tasks.

Artificial intelligence will play a crucial role in making things intelligent and achieving this goal. The most challenging aspect of robot navigation is path planning, which involves moving from one location to another while avoiding obstacles. The created agent (robot) can navigate the grid, track its state, and determine which path to take, including if it is in a goal state. Uninformed and Informed are search algorithms designed for this robot. Uninformed search methods include depth-first, breadth-first, Informed, Greedy uses best-first, A*, and modified search algorithms. Uniform Cost Search investigates the grid while accounting for the cost of accessing each node. It prioritizes paths with lower cumulative costs, assuring the best option in terms of path length.

Search Algorithms

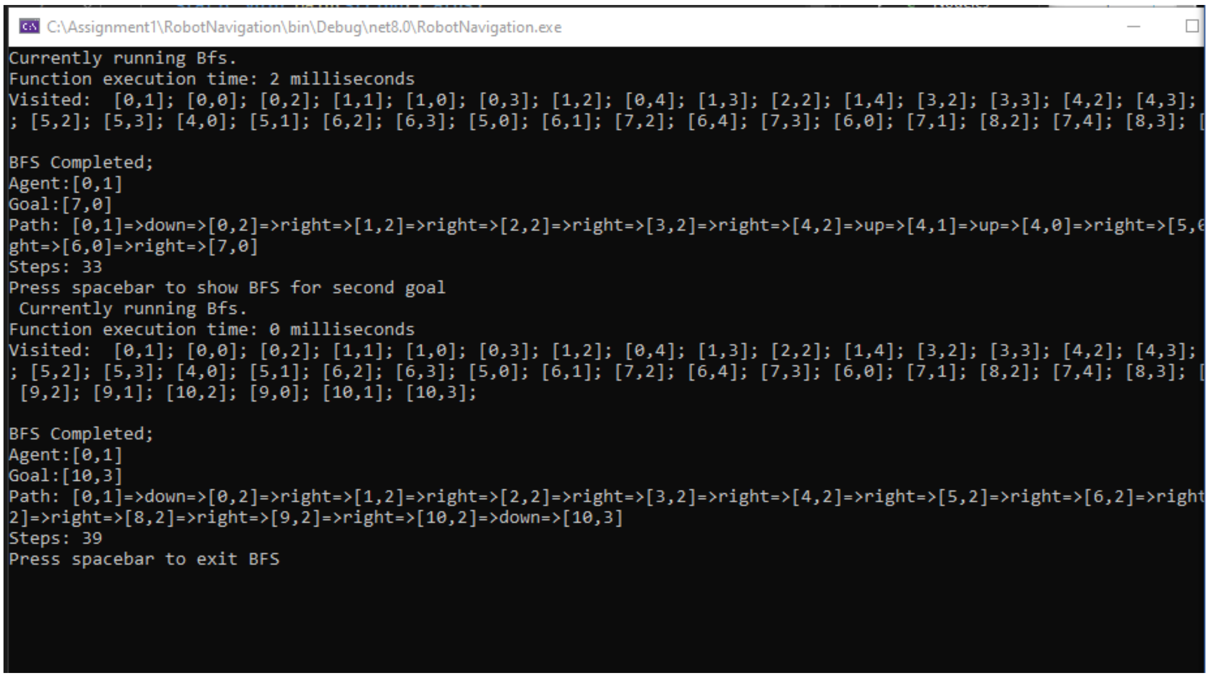
1) Breadth First Algorithm

Functionality: BFS begins from the agent's current location and proceeds to systematically investigate the grid, level by level. Before going farther into the search space, it considers every cell that is nearby at every level.

Execution

a) BFS keeps a queue for nodes that need to be investigated, which it refers to as the frontier.
b) The agent's starting point ,or root node, is first queued into frontier.
c) It repeatedly takes nodes out of the frontier queue and investigates the cells next to them.
d) BFS queues a neighboring cell for more investigation if it hasn't been visited, isn't a wall, and isn't in the frontier.
e) BFS keeps on this manner until the target cell is located or the The whole grid has been examined.

Termination - When the goal cell is located, BFS ends. At that point, it uses the parent pointers kept in each node to reconstruct. The path from the start to the goal. If BFS thoroughly searches the whole grid in search of the goal cell and is unable to locate one, it will produce a failure message.



```
C:\Assignment1\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe
Currently running Bfs.
Function execution time: 2 milliseconds
Visited:  [0,1]; [0,0]; [0,2]; [1,1]; [1,0]; [0,3]; [1,2]; [0,4]; [1,3]; [2,2]; [1,4]; [3,2]; [3,3]; [4,2]; [4,3];
; [5,2]; [5,3]; [4,0]; [5,1]; [6,2]; [6,3]; [5,0]; [6,1]; [7,2]; [6,4]; [7,3]; [6,0]; [7,1]; [8,2]; [7,4]; [8,3]; [

BFS Completed;
Agent:[0,1]
Goal:[7,0]
Path: [0,1]=>down=>[0,2]=>right=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>up=>[4,1]=>up=>[4,0]=>right=>[5,0
ght=>[6,0]=>right=>[7,0]
Steps: 33
Press spacebar to show BFS for second goal
 Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [0,1]; [0,0]; [0,2]; [1,1]; [1,0]; [0,3]; [1,2]; [0,4]; [1,3]; [2,2]; [1,4]; [3,2]; [3,3]; [4,2]; [4,3];
; [5,2]; [5,3]; [4,0]; [5,1]; [6,2]; [6,3]; [5,0]; [6,1]; [7,2]; [6,4]; [7,3]; [6,0]; [7,1]; [8,2]; [7,4]; [8,3]; [
 [9,2]; [9,1]; [10,2]; [9,0]; [10,1]; [10,3];

BFS Completed;
Agent:[0,1]
Goal:[10,3]
Path: [0,1]=>down=>[0,2]=>right=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>right=>[5,2]=>right=>[6,2]=>right
2]=>right=>[8,2]=>right=>[9,2]=>right=>[10,2]=>down=>[10,3]
Steps: 39
Press spacebar to exit BFS
```

Figure 4 Screenshot of BFS running.

2) <u>Depth First Search:</u>

Functionality- DFS's functionality entails travelling each branch of the grid as far as it can before turning around. It places more emphasis on depth than breadth, frequently delving thoroughly into one branch before examining other options.

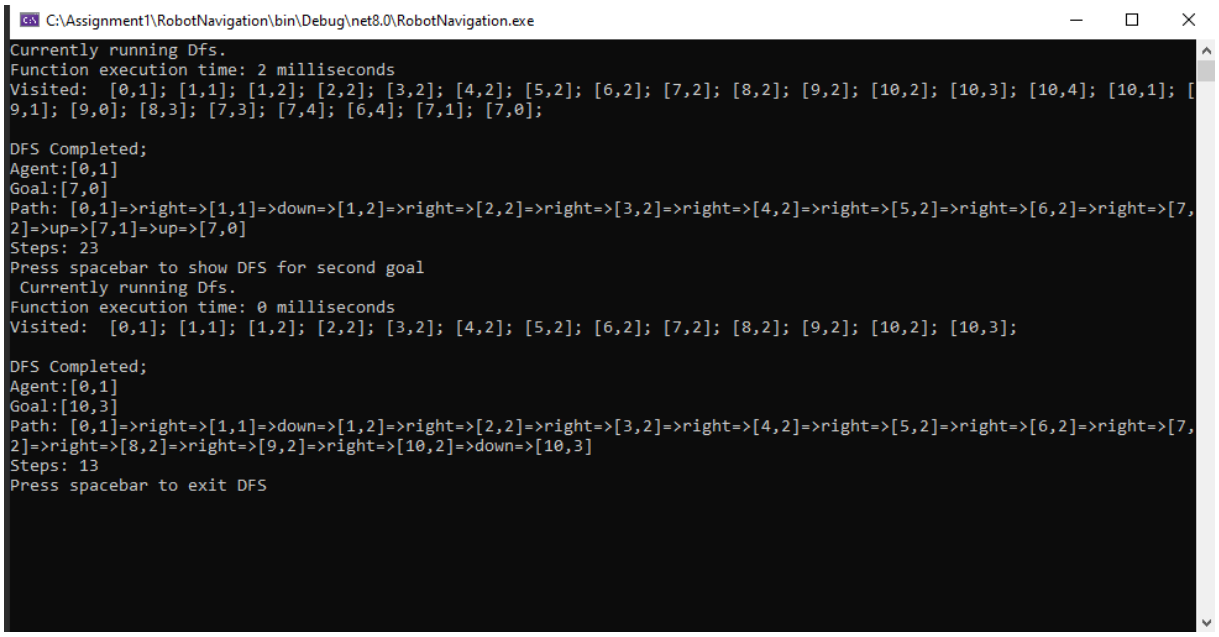
Execution:

- DFS keeps track of nodes that need to be investigated in a stack.
  it calls the frontier.
- The agent's starting location, or root node, is pushed onto the stack first.
- It removes nodes from the stack and investigates the cells next to them.
- DFS places a neighbouring cell onto the stack for additional investigation if it hasn't been visited, isn't a wall, and isn't already there.
- DFS keeps on in this manner until either the target cell is located, or every possible route has been taken.

Termination:

• After locating the goal cell, DFS stops and uses the parent pointers kept in each node to recreate the path from the start to the goal.

• DFS provides an error message stating that no solution could be found if it thoroughly searches every avenue without discovering the objective cell



C:\Assignment1\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

```
Currently running Dfs.
Function execution time: 2 milliseconds
Visited:  [0,1]; [1,1]; [1,2]; [2,2]; [3,2]; [4,2]; [5,2]; [6,2]; [7,2]; [8,2]; [9,2]; [10,2]; [10,3]; [10,4]; [10,1]; [
9,1]; [9,0]; [8,3]; [7,3]; [7,4]; [6,4]; [7,1]; [7,0];

DFS Completed;
Agent:[0,1]
Goal:[7,0]
Path: [0,1]=>right=>[1,1]=>down=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>right=>[5,2]=>right=>[6,2]=>right=>[7,
2]=>up=>[7,1]=>up=>[7,0]
Steps: 23
Press spacebar to show DFS for second goal
 Currently running Dfs.
Function execution time: 0 milliseconds
Visited:  [0,1]; [1,1]; [1,2]; [2,2]; [3,2]; [4,2]; [5,2]; [6,2]; [7,2]; [8,2]; [9,2]; [10,2]; [10,3];

DFS Completed;
Agent:[0,1]
Goal:[10,3]
Path: [0,1]=>right=>[1,1]=>down=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>right=>[5,2]=>right=>[6,2]=>right=>[7,
2]=>right=>[8,2]=>right=>[9,2]=>right=>[10,2]=>down=>[10,3]
Steps: 13
Press spacebar to exit DFS
```

Figure 5 Screenshot of DFS running

3) <u>**Uniform Cost Search:**</u>

Functionality- Equitable price when searching the grid , the cost of travelling to each node is considered. It guarantees the best outcome in terms of path length by giving priority to paths with lower cumulative costs.

Execution-

1) In order to store nodes in order of cumulative cost, Uniform Cost Search keeps track of a priority queue.

2) The root node, or the agent's starting location, is the first added to the priority queue at no cost.

3) It repeatedly takes nods out of the priority queue and investigates the cells next to them.

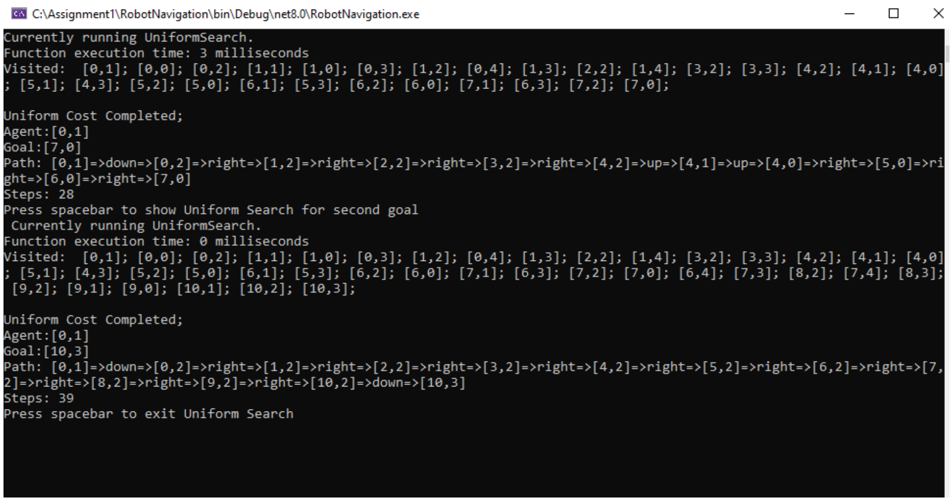
4) Uniform Cost Search determines a neighbour cell's cumulative cost from the start node and adds it to the priority queue if it hasn't been visited attend isn't a wall.

5) This procedure is carried out repeatedly using Uniform Cost Search until the target cell is located or all reachable nodes have been investigated.

Termination-

1) Using the parent pointers kept in each node, the Uniform Cost Search ends when the goal cell is located. This allows for the reconstruction of the best path from the start to the goal.

2) Uniform Cost Search produces a failure message stating that no solution could be discovered if it searches every reachable node without discovering the goal cell.


C:\Assignment1\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

```
Currently running UniformSearch.
Function execution time: 3 milliseconds
Visited:  [0,1]; [0,0]; [0,2]; [1,1]; [1,0]; [0,3]; [1,2]; [0,4]; [1,3]; [2,2]; [1,4]; [3,2]; [3,3]; [4,2]; [4,1]; [4,0]
; [5,1]; [4,3]; [5,2]; [5,0]; [6,1]; [5,3]; [6,2]; [6,0]; [7,1]; [6,3]; [7,2]; [7,0];

Uniform Cost Completed;
Agent:[0,1]
Goal:[7,0]
Path: [0,1]=>down=>[0,2]=>right=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>up=>[4,1]=>up=>[4,0]=>right=>[5,0]=>ri
ght=>[6,0]=>right=>[7,0]
Steps: 28
Press spacebar to show Uniform Search for second goal
 Currently running UniformSearch.
Function execution time: 0 milliseconds
Visited:  [0,1]; [0,0]; [0,2]; [1,1]; [1,0]; [0,3]; [1,2]; [0,4]; [1,3]; [2,2]; [1,4]; [3,2]; [3,3]; [4,2]; [4,1]; [4,0]
; [5,1]; [4,3]; [5,2]; [5,0]; [6,1]; [5,3]; [6,2]; [6,0]; [7,1]; [6,3]; [7,2]; [7,0]; [6,4]; [7,3]; [8,2]; [7,4]; [8,3];
 [9,2]; [9,1]; [9,0]; [10,1]; [10,2]; [10,3];

Uniform Cost Completed;
Agent:[0,1]
Goal:[10,3]
Path: [0,1]=>down=>[0,2]=>right=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>right=>[5,2]=>right=>[6,2]=>right=>[7,
2]=>right=>[8,2]=>right=>[9,2]=>right=>[10,2]=>down=>[10,3]
Steps: 39
Press spacebar to exit Uniform Search
```

Figure 6 Uniform Search Running

4) <u>**Greedy Best First Search**</u>:

Functionality- Minimise the estimated remaining distance to the objective by giving priority to nodes that are close to it using heuristic function.

Execution-

1) GBFS stores nodes in order of estimated distances to the objective in a priority queue, which is not explicitly evident in the code provided.

2) According to its heuristic value, it begins by adding the agent's starting position to the priority queue.

3) It repeatedly takes nodes out of the priority queue and investigates the cells next to them.

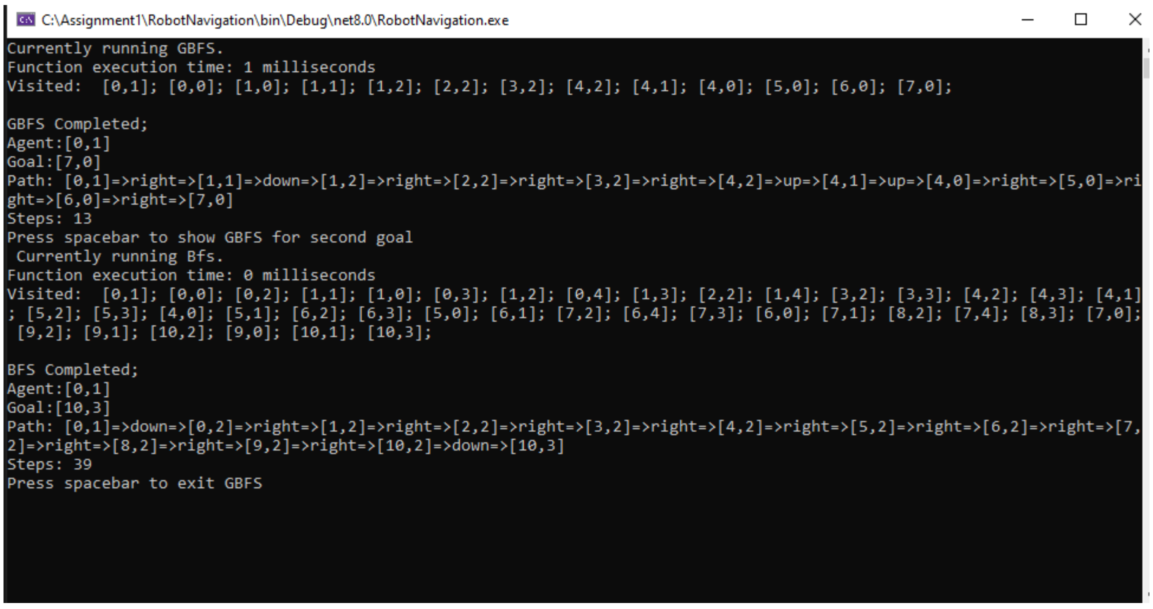
4) The neighbour cell that is closest to the target and has the lowest heuristic value is chosen by GBFS and is put in queue for additional investigation.

5) GBFS keeps going in this manner until either the target cell is located, or every reachable node has been investigated.

Termination-

1) When the goal cell is located, GBFS stops, using the parent pointers kept in each node to reconstruct the oath from the start to the goal.

2) When GBFS searches every reachable node in an attempt to locate the goal cell and fails, it provides a failure report stating that no solution was discovered.


C:\Assignment1\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

```
Currently running GBFS.
Function execution time: 1 milliseconds
Visited:  [0,1]; [0,0]; [1,0]; [1,1]; [1,2]; [2,2]; [3,2]; [4,2]; [4,1]; [4,0]; [5,0]; [6,0]; [7,0];

GBFS Completed;
Agent:[0,1]
Goal:[7,0]
Path: [0,1]=>right=>[1,1]=>down=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>up=>[4,1]=>up=>[4,0]=>right=>[5,0]=>ri
ght=>[6,0]=>right=>[7,0]
Steps: 13
Press spacebar to show GBFS for second goal
 Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [0,1]; [0,0]; [0,2]; [1,1]; [1,0]; [0,3]; [1,2]; [0,4]; [1,3]; [2,2]; [1,4]; [3,2]; [3,3]; [4,2]; [4,3]; [4,1]
; [5,2]; [5,3]; [4,0]; [5,1]; [6,2]; [6,3]; [5,0]; [6,1]; [7,2]; [6,4]; [7,3]; [6,0]; [7,1]; [8,2]; [7,4]; [8,3]; [7,0];
 [9,2]; [9,1]; [10,2]; [9,0]; [10,1]; [10,3];

BFS Completed;
Agent:[0,1]
Goal:[10,3]
Path: [0,1]=>down=>[0,2]=>right=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>right=>[5,2]=>right=>[6,2]=>right=>[7,
2]=>right=>[8,2]=>right=>[9,2]=>right=>[10,2]=>down=>[10,3]
Steps: 39
Press spacebar to exit GBFS
```

Figure 7 Screenshot of GBFS running.

5) **A- Star Search**

Functionality- By considering both the cost of arriving at a node and an estimate of the remaining cost to reach the objective, A*Search combines the advantages of uniform cost search with greedy best-first search. It seeks to identify the best course of action in terms of path length and heuristic estimation.

Execution-

1) In order to keep nodes in order of total estimated cost(the sum of the cost to reach the node and the heuristic estimate to reach the objective), A*Search maintains a priority queue, which is not directly visible in the code provided.

2) Based on its overall anticipated cost, it begins by adding the agent's starting position (root node) to the priority queue.

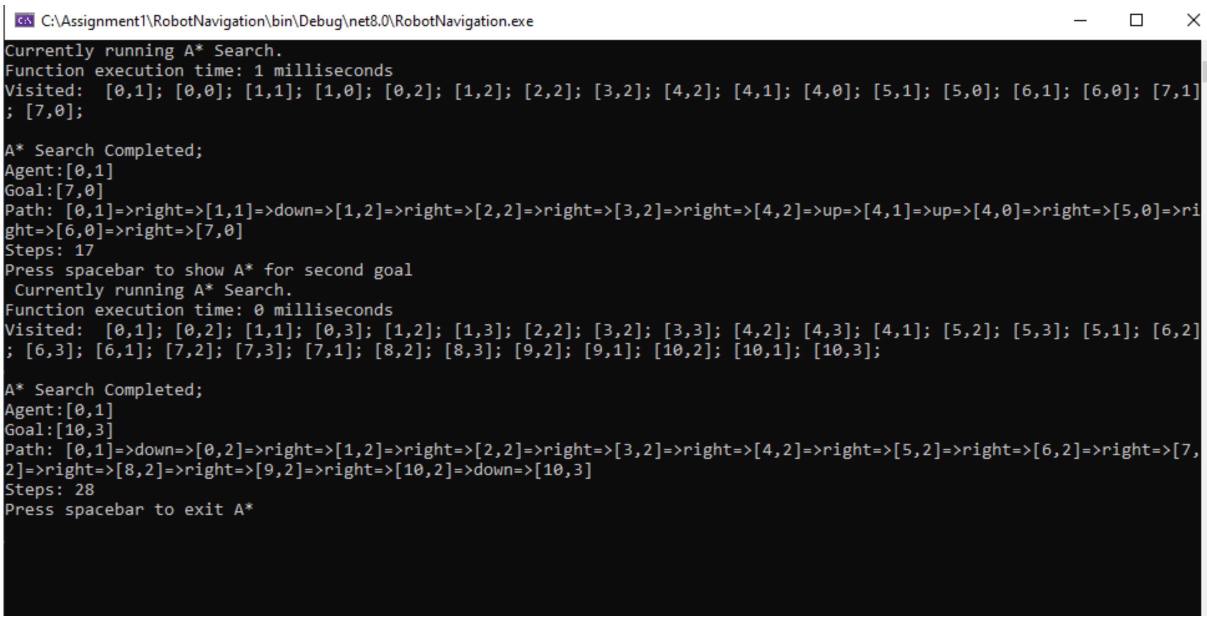
3) It repeatedly takes nodes out of the priority queue and investigates the cells next to them.

4) Every neighbouring cell's total estimated cost is determined using A*Search, which then adds it to the priority queue.

5) This process is repeated by A*Search until the target cell is located or until every reachable node has been investigated.

Termination-

1) When the goal cell is located, A* Search stops and uses the parent pointers that are saved in each node to reconstruct the best route from the start to the goal.
2) A*Search provides a failure message stating that no solution could be located if it searches every reachable node without discovering the goal cell.



```
C:\Assignment1\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe
Currently running A* Search.
Function execution time: 1 milliseconds
Visited:  [0,1]; [0,0]; [1,1]; [1,0]; [0,2]; [1,2]; [2,2]; [3,2]; [4,2]; [4,1]; [4,0]; [5,1]; [5,0]; [6,1]; [6,0]; [7,1]
; [7,0];

A* Search Completed;
Agent:[0,1]
Goal:[7,0]
Path: [0,1]=>right=>[1,1]=>down=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>up=>[4,1]=>up=>[4,0]=>right=>[5,0]=>ri
ght=>[6,0]=>right=>[7,0]
Steps: 17
Press spacebar to show A* for second goal
 Currently running A* Search.
Function execution time: 0 milliseconds
Visited:  [0,1]; [0,2]; [1,1]; [0,3]; [1,2]; [1,3]; [2,2]; [3,2]; [3,3]; [4,2]; [4,3]; [4,1]; [5,2]; [5,3]; [5,1]; [6,2]
; [6,3]; [6,1]; [7,2]; [7,3]; [7,1]; [8,2]; [8,3]; [9,2]; [9,1]; [10,2]; [10,1]; [10,3];

A* Search Completed;
Agent:[0,1]
Goal:[10,3]
Path: [0,1]=>down=>[0,2]=>right=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>right=>[5,2]=>right=>[6,2]=>right=>[7,
2]=>right=>[8,2]=>right=>[9,2]=>right=>[10,2]=>down=>[10,3]
Steps: 28
Press spacebar to exit A*
```

Figure 8 Screenshot of Astar search running.

6) **Dijkstra's Algorithm**

Functionality - Dijkstra's Algorithm, implemented in the Agent class, is a basic graph search algorithm that finds the shortest path from a source node to all other nodes in a weighted graph. Dijkstra's Algorithm works by iteratively selecting the node with the shortest tentative distance from the source and updating the distances to its neighbour nodes. It uses a priority queue (typically implemented as a min-heap) to efficiently select the next node to explore based on their estimated distance.

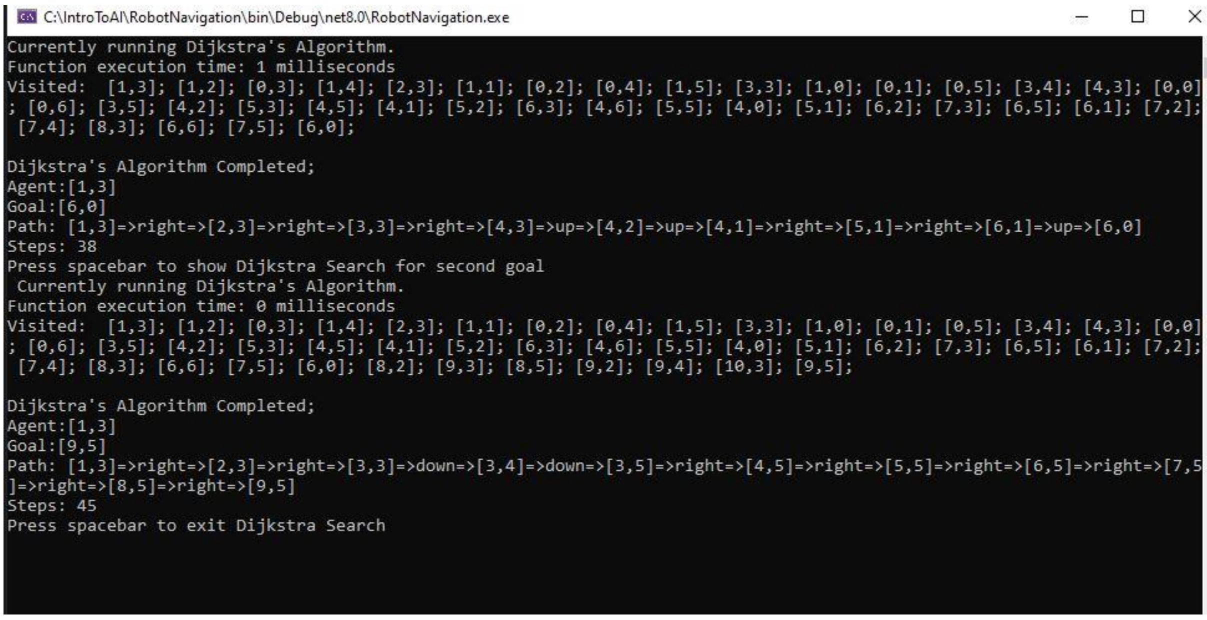
Implementation- The Agent class uses Dijkstra's Algorithm to discover the shortest path from the agent's present position to the goal point on a grid-based map. The method investigates neighbouring nodes in a systematic manner, taking into account the cost of accessing each node from its source.

Efficiency: Dijkstra's Algorithm guarantees the shortest path from the source to all subsequent nodes in the network, making it appropriate for cases requiring the best answer.

Completeness: The algorithm is complete, which means that it will always discover a solution if one exists, as long as the graph has no negative edge weights.

Optimality: Dijkstra's Algorithm ensures that the solution is optimal, as it finds the shortest path to each node.

Termination- Dijkstra's Algorithm is completed once all reachable nodes have been investigated and their shortest pathways from the source node established. This termination condition guarantees that the algorithm has thoroughly searched the graph and discovered the shortest paths to all reachable nodes.


C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

```
Currently running Dijkstra's Algorithm.
Function execution time: 1 milliseconds
Visited:  [1,3]; [1,2]; [0,3]; [1,4]; [2,3]; [1,1]; [0,2]; [0,4]; [1,5]; [3,3]; [1,0]; [0,1]; [0,5]; [3,4]; [4,3]; [0,0]
; [0,6]; [3,5]; [4,2]; [5,3]; [4,5]; [4,1]; [5,2]; [6,3]; [4,6]; [5,5]; [4,0]; [5,1]; [6,2]; [7,3]; [6,5]; [6,1]; [7,2];
 [7,4]; [8,3]; [6,6]; [7,5]; [6,0];

Dijkstra's Algorithm Completed;
Agent:[1,3]
Goal:[6,0]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>right=>[4,3]=>up=>[4,2]=>up=>[4,1]=>right=>[5,1]=>right=>[6,1]=>up=>[6,0]
Steps: 38
Press spacebar to show Dijkstra Search for second goal
 Currently running Dijkstra's Algorithm.
Function execution time: 0 milliseconds
Visited:  [1,3]; [1,2]; [0,3]; [1,4]; [2,3]; [1,1]; [0,2]; [0,4]; [1,5]; [3,3]; [1,0]; [0,1]; [0,5]; [3,4]; [4,3]; [0,0]
; [0,6]; [3,5]; [4,2]; [5,3]; [4,5]; [4,1]; [5,2]; [6,3]; [4,6]; [5,5]; [4,0]; [5,1]; [6,2]; [7,3]; [6,5]; [6,1]; [7,2];
 [7,4]; [8,3]; [6,6]; [7,5]; [6,0]; [8,2]; [9,3]; [8,5]; [9,2]; [9,4]; [10,3]; [9,5];

Dijkstra's Algorithm Completed;
Agent:[1,3]
Goal:[9,5]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>down=>[3,4]=>down=>[3,5]=>right=>[4,5]=>right=>[5,5]=>right=>[6,5]=>right=>[7,5
]=>right=>[8,5]=>right=>[9,5]
Steps: 45
Press spacebar to exit Dijkstra Search
```

Implementation

1) **Program.cs** - The primary point of entry for the program is the program class. A starter class instance called agent with the file path "test.txt" is created in the main method. The starting locator of the agent is indicated as 'A', while the goal position is indicated as 'G', indicating the initial state of the grid environment . The user is then presented with a selection of search algorithms that are available.

   After then, the application goes into a loop where it keeps asking the user for input until they decide to close it. The program uses the user's input to determine which search algorithm to run when it receives it. To give a clean interference , the console is cleaned before each search algorithm is run. The user is prompted to enter a valid response if they enter an invalid input.

   Every search algorithm, including BfsSearch, DfsSearch, GbfsSearch, AstarSearch, and UniformSearch has methods in the starter class. To determine the best route from the agent's starting point to the desired place, these techniques put the corresponding search algorithms into practice. Different paths are explored by search algorithm in the grid environment according to its unique strategy.

2) **Starter.cs -** The Starter class, which controls the entire search procedure, is the central component of the program. In addition to acting as the primary entry point , this class contains methods for configuring the search environment and running several search algorithms , including A*,Greedy Best-First(GBFS), Depth First Search (DFS), and Breadth-First Search(BFS). The Starter class creates a file reader object at instantiation, which oversees parsing and extracting crucial information from text files. This data includes important information including the size of the grid, the agent and goal placements, and the wall within the grid.

3) **Node.cs-** The Node C# class is used to represent individual nodes in a network that is utilized by tree-based search methods. The characteristics and actions of the nodes in the graph are contained in this class. The Node class has multiple private fields: edges to keep track of related nodes, parent to record the parent node, status to indicate whether the node is a wall, and x and y for the node's coordinates. To provide restricted access, certain fields are accessible via public spaces.

   There are three constructors available in class:

   - A constructor without parameters sets a node's initial values.
   - A constructor initializes the node based on its coordinates (x and y) which it receives.
   - There is another constructor that takes a parent node in addition to initializing the node with the provided coordinates.

4) **FileReader.cs-** The file reader class that may be used to parse and extract data from a text file. This class is designed to meet the requirements of a tree-based search problem set in a grid context.

Fundamentally, the file reader class has constructor, fields, and methods designed to manage the parsing and extraction of relevant data from the input text file.

- Declarations in field: Different data piece extracted from the text file are represented by the lines, grid, agentops, goal, and walls. Reading data from the file is facilitated by a stream reader object called reader.
- Builder: The constructor initializes a streamReader object to open and read from the given file it receives a filename as an input.

Techniques:
a) Read: The approach traverses every line in the input file iteratively, processing and allocating data to pertinent fields according to their locations in the file. It recognises and extracts data about the size of the grid, the position of the agent, the goal, and the locations of the walls.

b) GetAgent, GetGoal, and GetGrid: These techniques offer structured access to this crucial data by parsing the string representations of target locations, grid size ,and agent position respectively , into lists of integers.

c) GetWall: this function manages the parsing of lines that indicate the locations of walls in the grid. The method divides every line into whole numbers and arranges them into lists, providing a systematic depiction of wall locations for additional manipulation.

5) Agent.cs-

Fields and Constructor: The class contains fields that record information about the search problem, such as the root node, goal coordinates, grid dimensions, and walls. These fields are initialised by the constructor using the parameters sent to it.

Search Methods: Each search technique is implemented separately, including BFS, DFS, Uniform Cost Search, Greedy Best-First Search, and A* Search. Each approach has a similar structure of initialising data structures (e.g., queues, stacks, lists) and iteratively searching until the goal is accomplished or no solution is found.

Helper Methods: Several private methods are defined to help with the search process, including adding adjacent nodes to the frontier, creating the path string, and printing execution time.

Function AddAdjacentNodesToList

1) Loop, through the neighboring cells (up left right) of the node.
2) Generate nodes for positions within the grid boundaries.
3) Use AddNodeToFrontier to decide if nodes qualify and include them in the frontier.
4) Ensure that neighboring nodes are not walls have not been visited before. Are not already part of the frontier.

Function AddNodeToFrontier:

1) Verify if a node meets the conditions to be added to the frontier.
2) Check that the node is not already in the frontier is not the starting node has not been visited before and is not a wall.
3) Optimize traversal, by avoiding revisiting nodes that have already been explored or blocked.

<u>Pathfinding and Output:</u> The class uses the chosen algorithm to generate pathways from the starting place to the destination. It provides pertinent information, such as execution duration, visited nodes, path details, and the number of steps taken, to aid in the monitoring and analysis of algorithm behaviour.

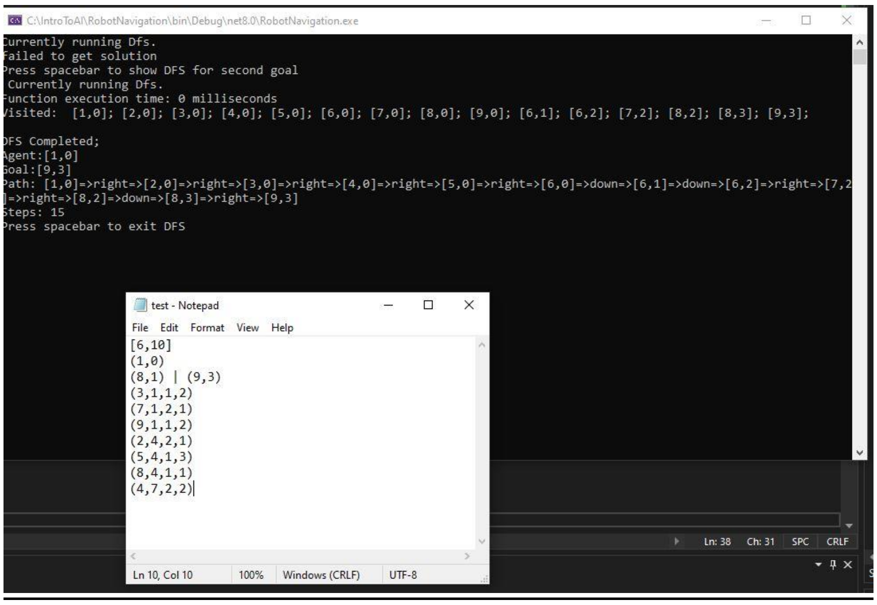
<u>Testing and Validation:</u> Thorough testing across multiple scenarios, such as varied grid configurations and obstacle layouts, is performed to confirm the validity and resilience of the implemented algorithms. This ensures consistent performance across a variety of pathfinding circumstances.

Testing

These test cases examine an agent's ability in navigating a grid-based environment with obstacles represented by walls.

Ten test cases were created to simulate diverse scenarios with variable grid sizes, beginning positions for the agent, goal placements, and wall configurations. Each test case has the following components:

Grid Size: The size of the grid dictate the spatial limitations under which the agent functions.
Agent Position: The agent's starting position within the grid.
Goal States: The positions that the agent must attain in order to effectively execute the assignment. Certain test cases may include many goal states.
Walls are rectangular obstacles that are identified by their position (top-left corner coordinates), width, and height.



C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

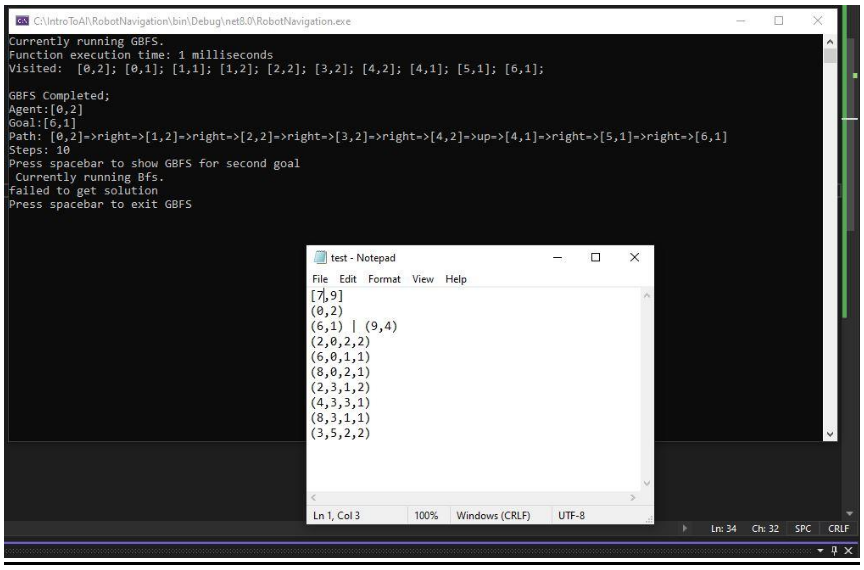
```
urrently running Dfs.
ailed to get solution
ress spacebar to show DFS for second goal
 Currently running Dfs.
unction execution time: 0 milliseconds
isited:  [1,0]; [2,0]; [3,0]; [4,0]; [5,0]; [6,0]; [7,0]; [8,0]; [9,0]; [6,1]; [6,2]; [7,2]; [8,2]; [8,3]; [9,3];

FS Completed;
gent:[1,0]
oal:[9,3]
ath: [1,0]=>right=>[2,0]=>right=>[3,0]=>right=>[4,0]=>right=>[5,0]=>right=>[6,0]=>down=>[6,1]=>down=>[6,2]=>right=>[7,2
]=>right=>[8,2]=>down=>[8,3]=>right=>[9,3]
teps: 15
ress spacebar to exit DFS
```

test - Notepad

```
[6,10]
(1,0)
(8,1) | (9,3)
(3,1,1,2)
(7,1,2,1)
(9,1,1,2)
(2,4,2,1)
(5,4,1,3)
(8,4,1,1)
(4,7,2,2)
```

Ln 10, Col 10 | 100% | Windows (CRLF) | UTF-8

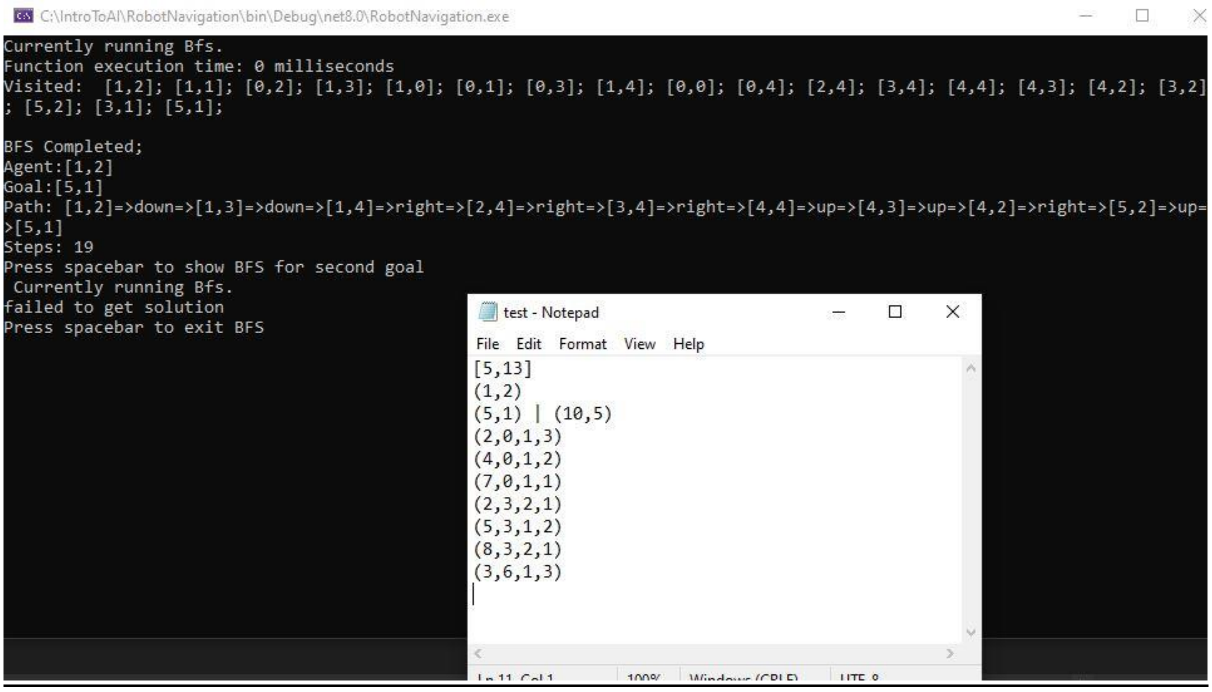
Figure 9 Test 1



```
Currently running GBFS.
Function execution time: 1 milliseconds
Visited:  [0,2]; [0,1]; [1,1]; [1,2]; [2,2]; [3,2]; [4,2]; [4,1]; [5,1]; [6,1];

GBFS Completed;
Agent:[0,2]
Goal:[6,1]
Path: [0,2]=>right=>[1,2]=>right=>[2,2]=>right=>[3,2]=>right=>[4,2]=>up=>[4,1]=>right=>[5,1]=>right=>[6,1]
Steps: 10
Press spacebar to show GBFS for second goal
 Currently running Bfs.
failed to get solution
Press spacebar to exit GBFS
```

```
[7,9]
(0,2)
(6,1) | (9,4)
(2,0,2,2)
(6,0,1,1)
(8,0,2,1)
(2,3,1,2)
(4,3,3,1)
(8,3,1,1)
(3,5,2,2)
```

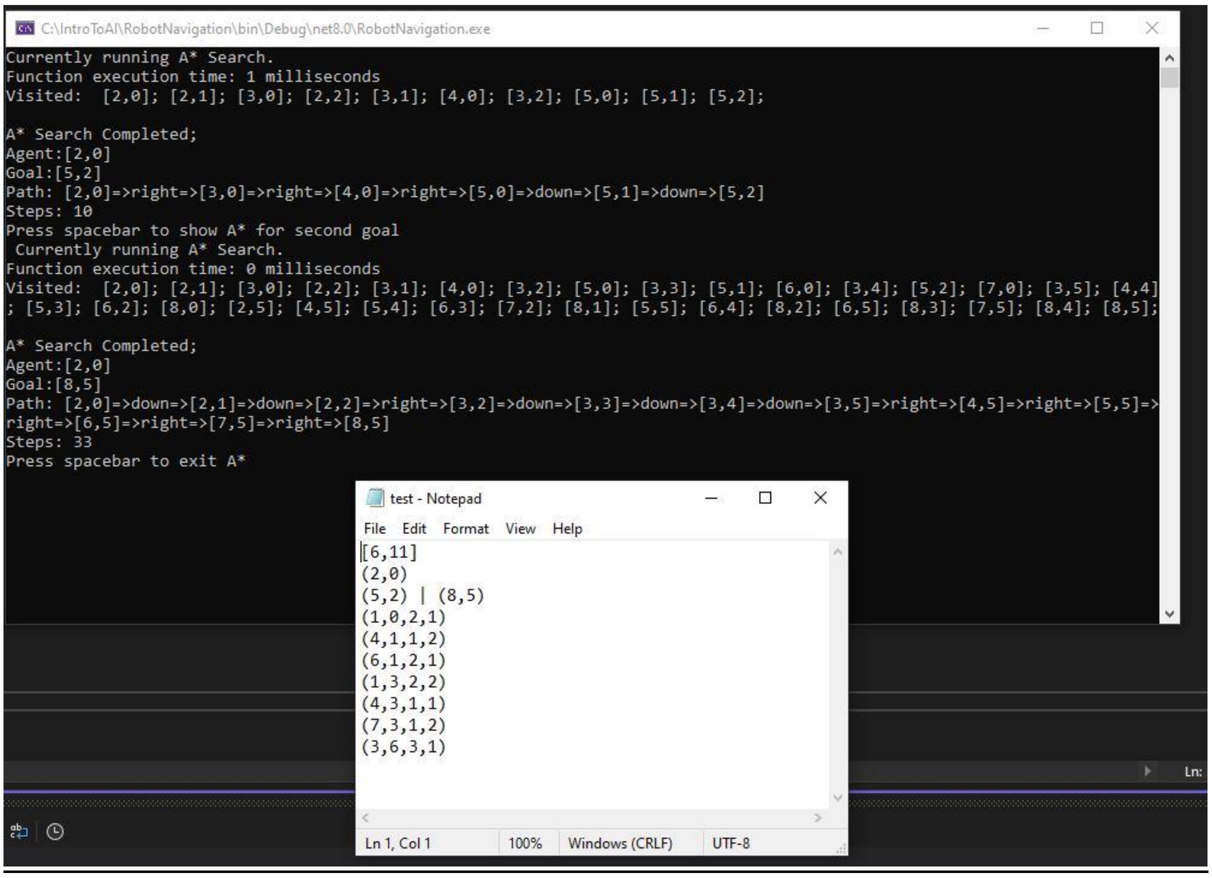
Figure 10 Test 2



```
Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [1,2]; [1,1]; [0,2]; [1,3]; [1,0]; [0,1]; [0,3]; [1,4]; [0,0]; [0,4]; [2,4]; [3,4]; [4,4]; [4,3]; [4,2]; [3,2]
; [5,2]; [3,1]; [5,1];

BFS Completed;
Agent:[1,2]
Goal:[5,1]
Path: [1,2]=>down=>[1,3]=>down=>[1,4]=>right=>[2,4]=>right=>[3,4]=>right=>[4,4]=>up=>[4,3]=>up=>[4,2]=>right=>[5,2]=>up=
>[5,1]
Steps: 19
Press spacebar to show BFS for second goal
 Currently running Bfs.
failed to get solution
Press spacebar to exit BFS
```

```
[5,13]
(1,2)
(5,1) | (10,5)
(2,0,1,3)
(4,0,1,2)
(7,0,1,1)
(2,3,2,1)
(5,3,1,2)
(8,3,2,1)
(3,6,1,3)
```

Figure 11 Test 3


C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

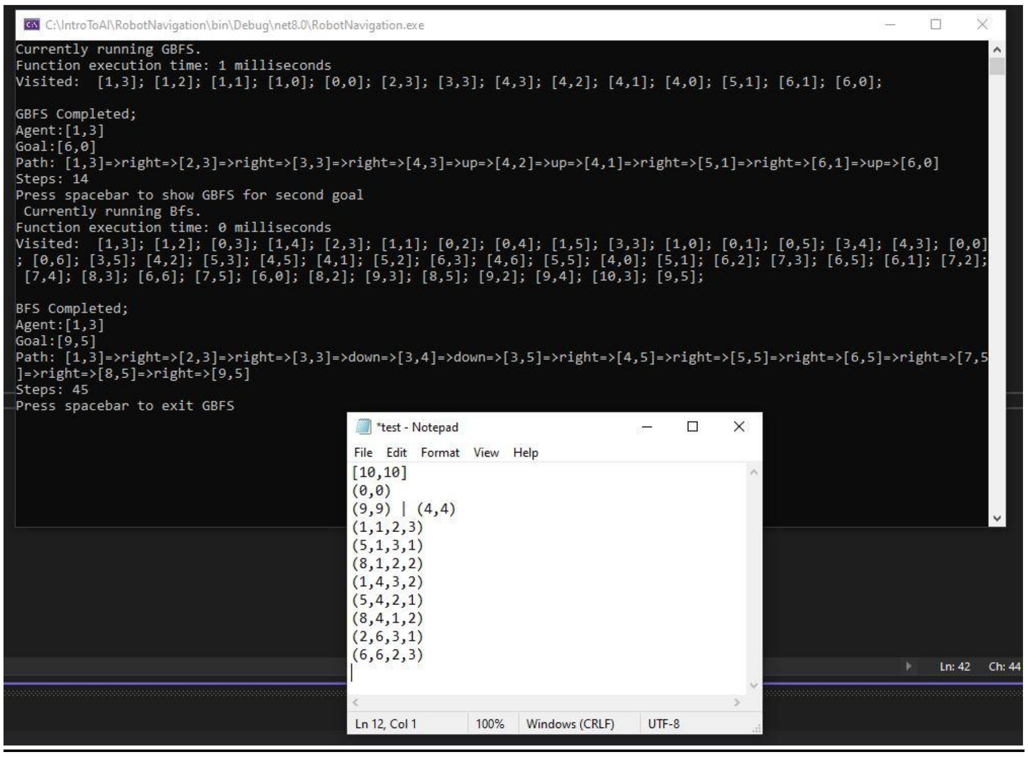
```
Currently running A* Search.
Function execution time: 1 milliseconds
Visited:  [2,0]; [2,1]; [3,0]; [2,2]; [3,1]; [4,0]; [3,2]; [5,0]; [5,1]; [5,2];

A* Search Completed;
Agent:[2,0]
Goal:[5,2]
Path: [2,0]=>right=>[3,0]=>right=>[4,0]=>right=>[5,0]=>down=>[5,1]=>down=>[5,2]
Steps: 10
Press spacebar to show A* for second goal
 Currently running A* Search.
Function execution time: 0 milliseconds
Visited:  [2,0]; [2,1]; [3,0]; [2,2]; [3,1]; [4,0]; [3,2]; [5,0]; [3,3]; [5,1]; [6,0]; [3,4]; [5,2]; [7,0]; [3,5]; [4,4]
; [5,3]; [6,2]; [8,0]; [2,5]; [4,5]; [5,4]; [6,3]; [7,2]; [8,1]; [5,5]; [6,4]; [8,2]; [6,5]; [8,3]; [7,5]; [8,4]; [8,5];

A* Search Completed;
Agent:[2,0]
Goal:[8,5]
Path: [2,0]=>down=>[2,1]=>down=>[2,2]=>right=>[3,2]=>down=>[3,3]=>down=>[3,4]=>down=>[3,5]=>right=>[4,5]=>right=>[5,5]=>
right=>[6,5]=>right=>[7,5]=>right=>[8,5]
Steps: 33
Press spacebar to exit A*
```

test - Notepad

```
[6,11]
(2,0)
(5,2) | (8,5)
(1,0,2,1)
(4,1,1,2)
(6,1,2,1)
(1,3,2,2)
(4,3,1,1)
(7,3,1,2)
(3,6,3,1)
```

Figure 12 Test 4

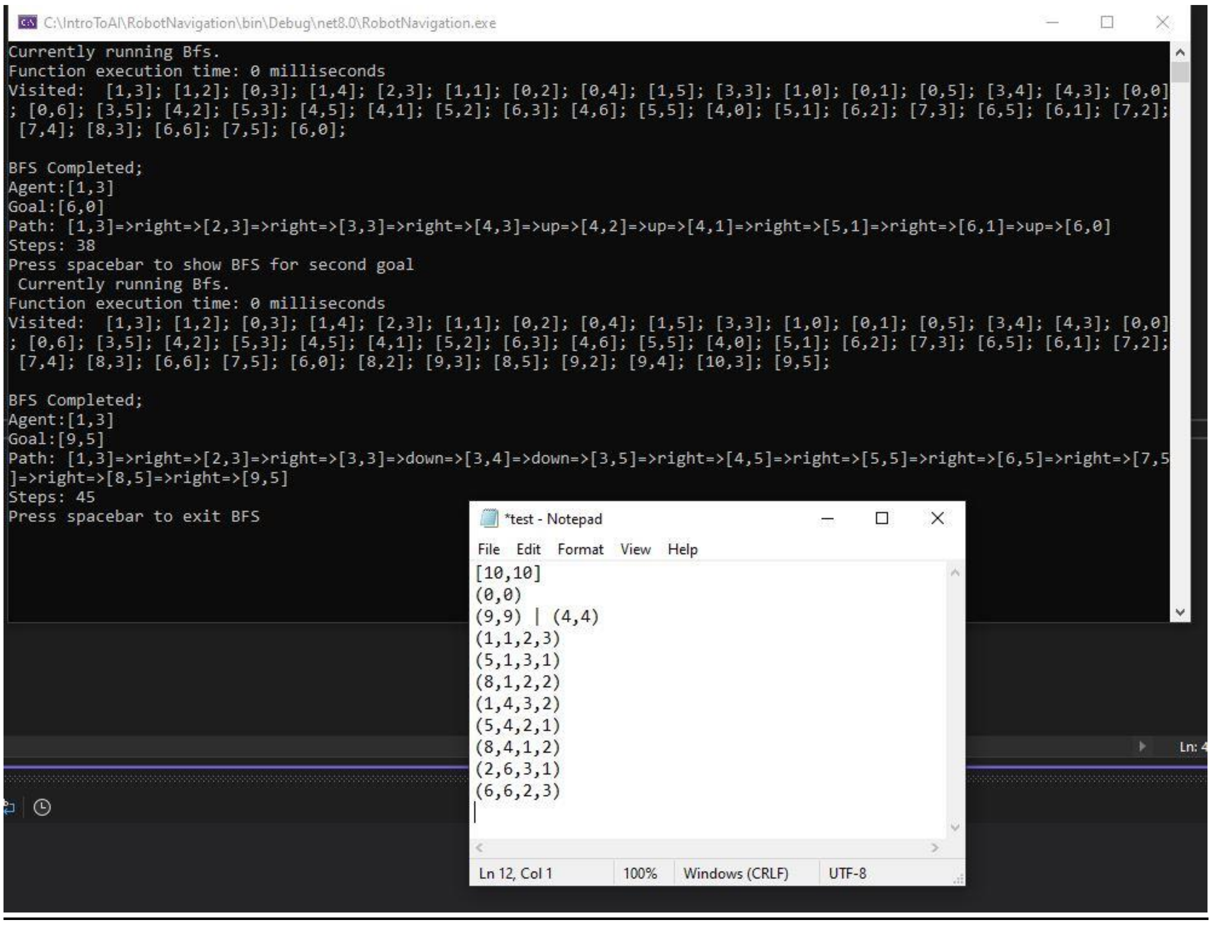
I did extensive testing on my search methods using bigger grid sizes and making the navigation more difficult for the agent as shown in the examples below. Also I tried to add GUI to make the program more attractive and give users a better experience in running the code. But unfortunately I wasn't able to implement the GUI features in this assignment. But I learned a lot about the vectors 2D and 3D that I coudve used in the program.



```
C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe
Currently running GBFS.
Function execution time: 1 milliseconds
Visited:  [1,3]; [1,2]; [1,1]; [1,0]; [0,0]; [2,3]; [3,3]; [4,3]; [4,2]; [4,1]; [4,0]; [5,1]; [6,1]; [6,0];

GBFS Completed;
Agent:[1,3]
Goal:[6,0]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>right=>[4,3]=>up=>[4,2]=>up=>[4,1]=>right=>[5,1]=>right=>[6,1]=>up=>[6,0]
Steps: 14
Press spacebar to show GBFS for second goal
 Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [1,3]; [1,2]; [0,3]; [1,4]; [2,3]; [1,1]; [0,2]; [0,4]; [1,5]; [3,3]; [1,0]; [0,1]; [0,5]; [3,4]; [4,3]; [0,0]
; [0,6]; [3,5]; [4,2]; [5,3]; [4,5]; [4,1]; [5,2]; [6,3]; [4,6]; [5,5]; [4,0]; [5,1]; [6,2]; [7,3]; [6,5]; [6,1]; [7,2];
 [7,4]; [8,3]; [6,6]; [7,5]; [6,0]; [8,2]; [9,3]; [8,5]; [9,2]; [9,4]; [10,3]; [9,5];

BFS Completed;
Agent:[1,3]
Goal:[9,5]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>down=>[3,4]=>down=>[3,5]=>right=>[4,5]=>right=>[5,5]=>right=>[6,5]=>right=>[7,5
]=>right=>[8,5]=>right=>[9,5]
Steps: 45
Press spacebar to exit GBFS
```

```
*test - Notepad
[10,10]
(0,0)
(9,9) | (4,4)
(1,1,2,3)
(5,1,3,1)
(8,1,2,2)
(1,4,3,2)
(5,4,2,1)
(8,4,1,2)
(2,6,3,1)
(6,6,2,3)

Ln 12, Col 1    100%    Windows (CRLF)    UTF-8
```

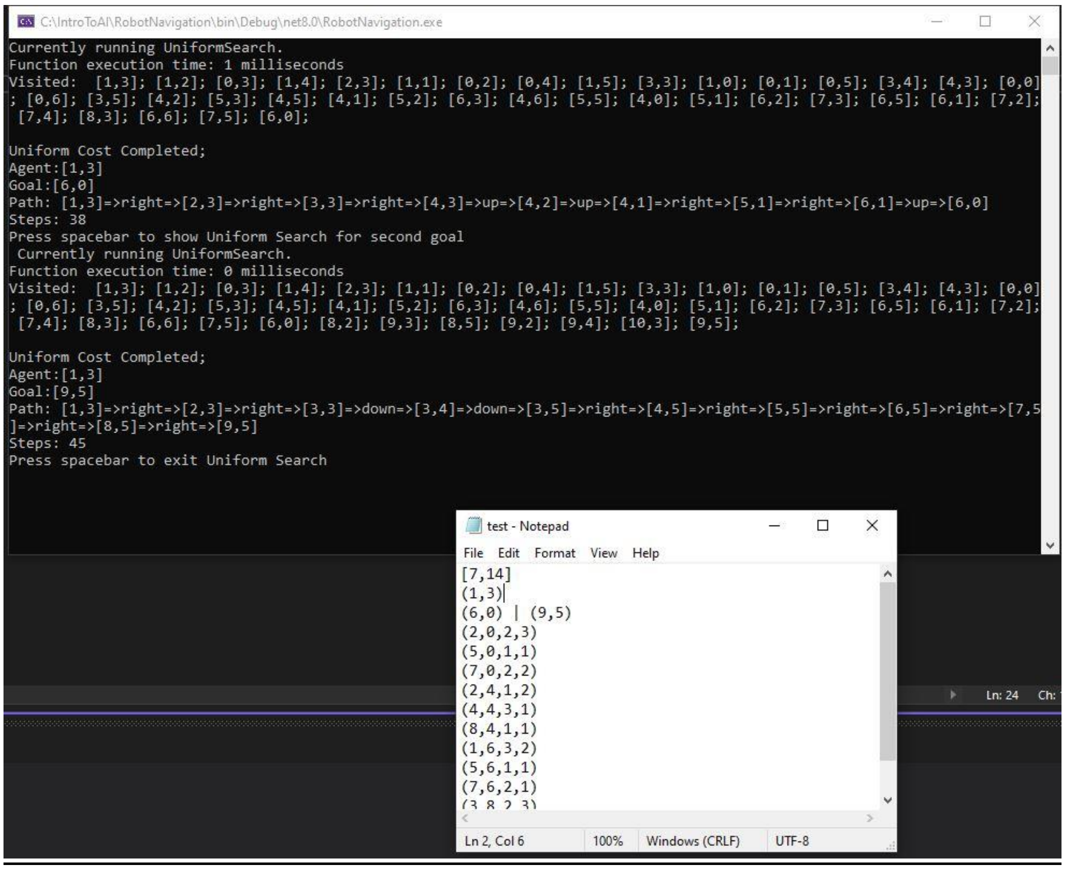
Figure 13 Test 5


C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

```
Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [1,3]; [1,2]; [0,3]; [1,4]; [2,3]; [1,1]; [0,2]; [0,4]; [1,5]; [3,3]; [1,0]; [0,1]; [0,5]; [3,4]; [4,3]; [0,0]
; [0,6]; [3,5]; [4,2]; [5,3]; [4,5]; [4,1]; [5,2]; [6,3]; [4,6]; [5,5]; [4,0]; [5,1]; [6,2]; [7,3]; [6,5]; [6,1]; [7,2];
 [7,4]; [8,3]; [6,6]; [7,5]; [6,0];

BFS Completed;
Agent:[1,3]
Goal:[6,0]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>right=>[4,3]=>up=>[4,2]=>up=>[4,1]=>right=>[5,1]=>right=>[6,1]=>up=>[6,0]
Steps: 38
Press spacebar to show BFS for second goal
 Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [1,3]; [1,2]; [0,3]; [1,4]; [2,3]; [1,1]; [0,2]; [0,4]; [1,5]; [3,3]; [1,0]; [0,1]; [0,5]; [3,4]; [4,3]; [0,0]
; [0,6]; [3,5]; [4,2]; [5,3]; [4,5]; [4,1]; [5,2]; [6,3]; [4,6]; [5,5]; [4,0]; [5,1]; [6,2]; [7,3]; [6,5]; [6,1]; [7,2];
 [7,4]; [8,3]; [6,6]; [7,5]; [6,0]; [8,2]; [9,3]; [8,5]; [9,2]; [9,4]; [10,3]; [9,5];

BFS Completed;
Agent:[1,3]
Goal:[9,5]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>down=>[3,4]=>down=>[3,5]=>right=>[4,5]=>right=>[5,5]=>right=>[6,5]=>right=>[7,5
]=>right=>[8,5]=>right=>[9,5]
Steps: 45
Press spacebar to exit BFS
```

```
[10,10]
(0,0)
(9,9) | (4,4)
(1,1,2,3)
(5,1,3,1)
(8,1,2,2)
(1,4,3,2)
(5,4,2,1)
(8,4,1,2)
(2,6,3,1)
(6,6,2,3)
```

Figure 14 Test 6



C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

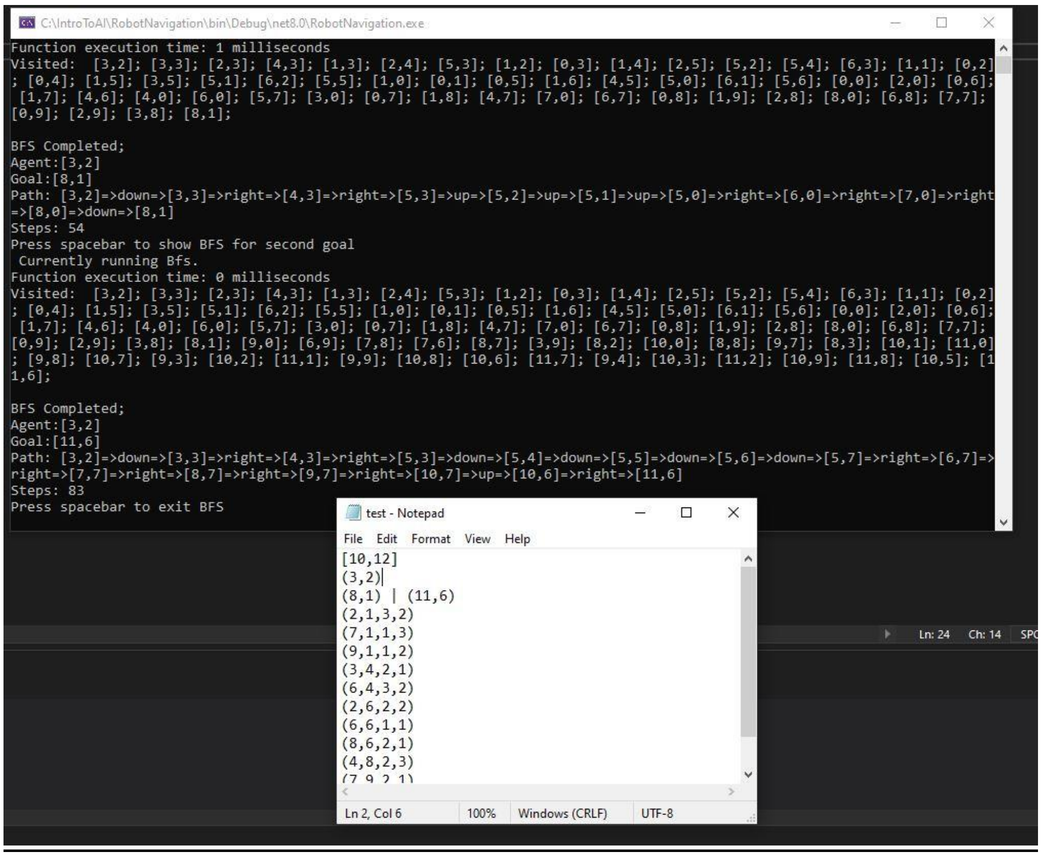
```
Currently running UniformSearch.
Function execution time: 1 milliseconds
Visited:  [1,3]; [1,2]; [0,3]; [1,4]; [2,3]; [1,1]; [0,2]; [0,4]; [1,5]; [3,3]; [1,0]; [0,1]; [0,5]; [3,4]; [4,3]; [0,0]
; [0,6]; [3,5]; [4,2]; [5,3]; [4,5]; [4,1]; [5,2]; [6,3]; [4,6]; [5,5]; [4,0]; [5,1]; [6,2]; [7,3]; [6,5]; [6,1]; [7,2];
 [7,4]; [8,3]; [6,6]; [7,5]; [6,0];

Uniform Cost Completed;
Agent:[1,3]
Goal:[6,0]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>right=>[4,3]=>up=>[4,2]=>up=>[4,1]=>right=>[5,1]=>right=>[6,1]=>up=>[6,0]
Steps: 38
Press spacebar to show Uniform Search for second goal
 Currently running UniformSearch.
Function execution time: 0 milliseconds
Visited:  [1,3]; [1,2]; [0,3]; [1,4]; [2,3]; [1,1]; [0,2]; [0,4]; [1,5]; [3,3]; [1,0]; [0,1]; [0,5]; [3,4]; [4,3]; [0,0]
; [0,6]; [3,5]; [4,2]; [5,3]; [4,5]; [4,1]; [5,2]; [6,3]; [4,6]; [5,5]; [4,0]; [5,1]; [6,2]; [7,3]; [6,5]; [6,1]; [7,2];
 [7,4]; [8,3]; [6,6]; [7,5]; [6,0]; [8,2]; [9,3]; [8,5]; [9,2]; [9,4]; [10,3]; [9,5];

Uniform Cost Completed;
Agent:[1,3]
Goal:[9,5]
Path: [1,3]=>right=>[2,3]=>right=>[3,3]=>down=>[3,4]=>down=>[3,5]=>right=>[4,5]=>right=>[5,5]=>right=>[6,5]=>right=>[7,5
]=>right=>[8,5]=>right=>[9,5]
Steps: 45
Press spacebar to exit Uniform Search
```

test - Notepad

```
[7,14]
(1,3)
(6,0) | (9,5)
(2,0,2,3)
(5,0,1,1)
(7,0,2,2)
(2,4,1,2)
(4,4,3,1)
(8,4,1,1)
(1,6,3,2)
(5,6,1,1)
(7,6,2,1)
```

Ln 2, Col 6 100% Windows (CRLF) UTF-8

Figure 15 Test 7



C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe

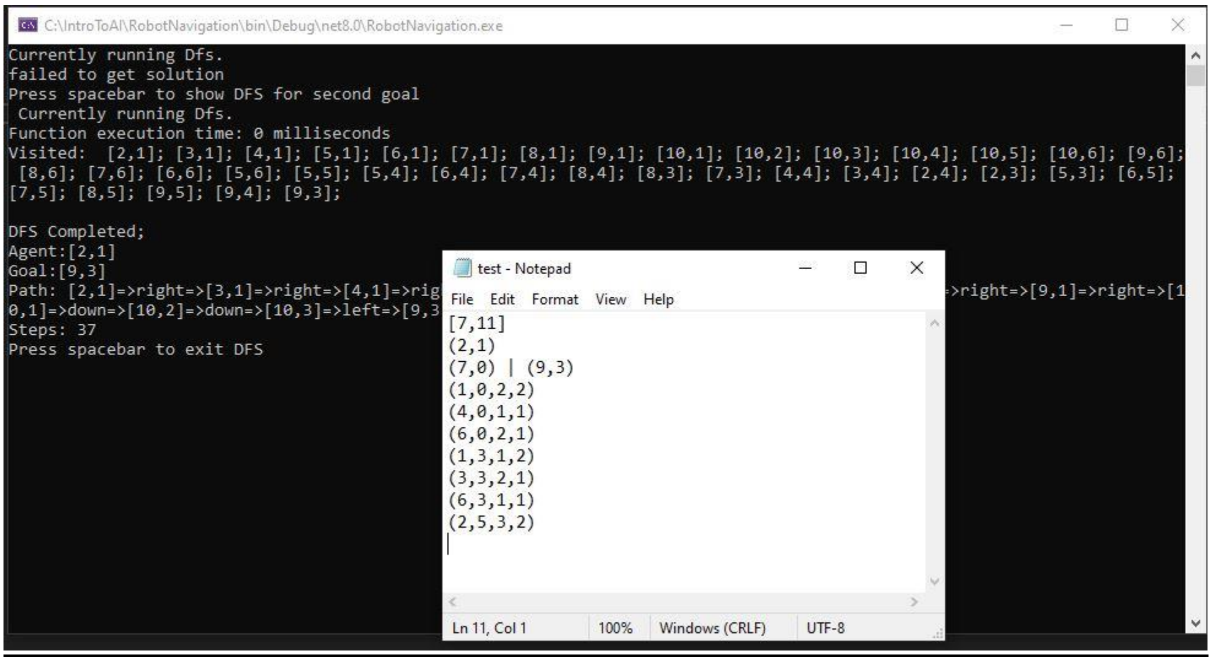
```
Function execution time: 1 milliseconds
Visited:  [3,2]; [3,3]; [2,3]; [4,3]; [1,3]; [2,4]; [5,3]; [1,2]; [0,3]; [1,4]; [2,5]; [5,2]; [5,4]; [6,3]; [1,1]; [0,2]
; [0,4]; [1,5]; [3,5]; [5,1]; [6,2]; [5,5]; [1,0]; [0,1]; [0,5]; [1,6]; [4,5]; [5,0]; [6,1]; [5,6]; [0,0]; [2,0]; [0,6];
 [1,7]; [4,6]; [4,0]; [6,0]; [5,7]; [3,0]; [0,7]; [1,8]; [4,7]; [7,0]; [6,7]; [0,8]; [1,9]; [2,8]; [8,0]; [6,8]; [7,7];
[0,9]; [2,9]; [3,8]; [8,1];

BFS Completed;
Agent:[3,2]
Goal:[8,1]
Path: [3,2]=>down=>[3,3]=>right=>[4,3]=>right=>[5,3]=>up=>[5,2]=>up=>[5,1]=>up=>[5,0]=>right=>[6,0]=>right=>[7,0]=>right
=>[8,0]=>down=>[8,1]
Steps: 54
Press spacebar to show BFS for second goal
 Currently running Bfs.
Function execution time: 0 milliseconds
Visited:  [3,2]; [3,3]; [2,3]; [4,3]; [1,3]; [2,4]; [5,3]; [1,2]; [0,3]; [1,4]; [2,5]; [5,2]; [5,4]; [6,3]; [1,1]; [0,2]
; [0,4]; [1,5]; [3,5]; [5,1]; [6,2]; [5,5]; [1,0]; [0,1]; [0,5]; [1,6]; [4,5]; [5,0]; [6,1]; [5,6]; [0,0]; [2,0]; [0,6];
 [1,7]; [4,6]; [4,0]; [6,0]; [5,7]; [3,0]; [0,7]; [1,8]; [4,7]; [7,0]; [6,7]; [0,8]; [1,9]; [2,8]; [8,0]; [6,8]; [7,7];
[0,9]; [2,9]; [3,8]; [8,1]; [9,0]; [6,9]; [7,8]; [7,6]; [8,7]; [3,9]; [8,2]; [10,0]; [8,8]; [9,7]; [8,3]; [10,1]; [11,0]
; [9,8]; [10,7]; [9,3]; [10,2]; [11,1]; [9,9]; [10,8]; [10,6]; [11,7]; [9,4]; [10,3]; [11,2]; [10,9]; [11,8]; [10,5]; [1
1,6];

BFS Completed;
Agent:[3,2]
Goal:[11,6]
Path: [3,2]=>down=>[3,3]=>right=>[4,3]=>right=>[5,3]=>down=>[5,4]=>down=>[5,5]=>down=>[5,6]=>down=>[5,7]=>right=>[6,7]=>
right=>[7,7]=>right=>[8,7]=>right=>[9,7]=>right=>[10,7]=>up=>[10,6]=>right=>[11,6]
Steps: 83
Press spacebar to exit BFS
```

test - Notepad

```
[10,12]
(3,2)
(8,1) | (11,6)
(2,1,3,2)
(7,1,1,3)
(9,1,1,2)
(3,4,2,1)
(6,4,3,2)
(2,6,2,2)
(6,6,1,1)
(8,6,2,1)
(4,8,2,3)
```

Ln 2, Col 6 100% Windows (CRLF) UTF-8

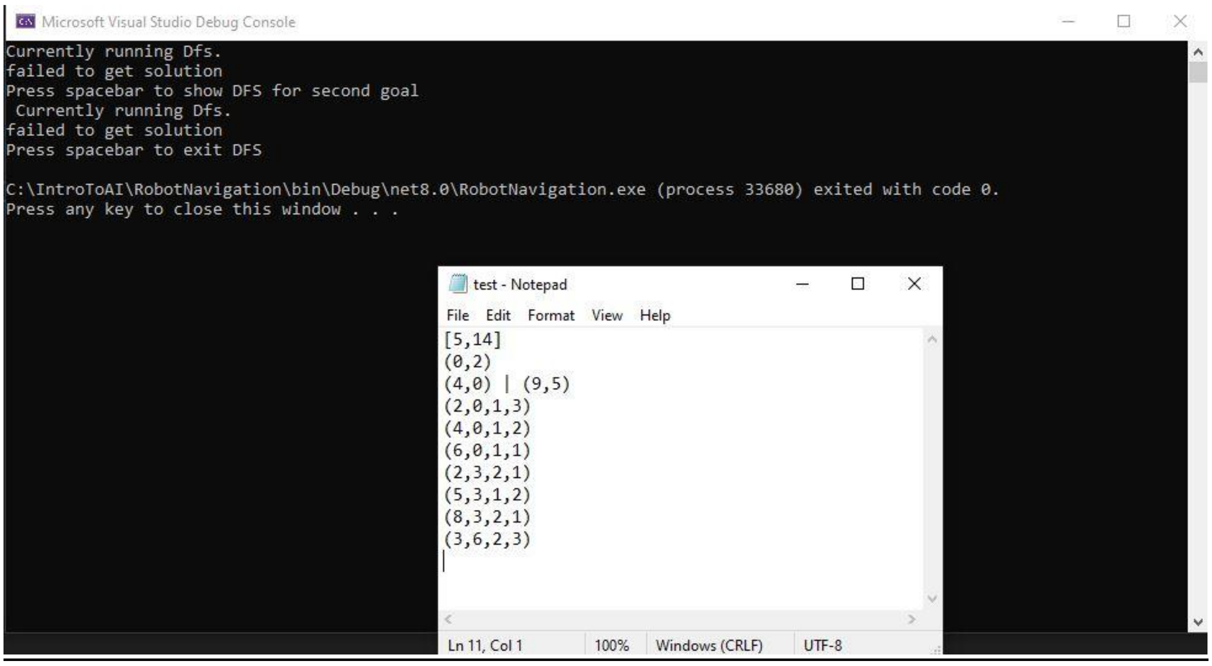
Figure 16 Test 8



```
C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe
Currently running Dfs.
failed to get solution
Press spacebar to show DFS for second goal
 Currently running Dfs.
Function execution time: 0 milliseconds
Visited:  [2,1]; [3,1]; [4,1]; [5,1]; [6,1]; [7,1]; [8,1]; [9,1]; [10,1]; [10,2]; [10,3]; [10,4]; [10,5]; [10,6]; [9,6];
 [8,6]; [7,6]; [6,6]; [5,6]; [5,5]; [5,4]; [6,4]; [7,4]; [8,4]; [8,3]; [7,3]; [4,4]; [3,4]; [2,4]; [2,3]; [5,3]; [6,5];
[7,5]; [8,5]; [9,5]; [9,4]; [9,3];

DFS Completed;
Agent:[2,1]
Goal:[9,3]
Path: [2,1]=>right=>[3,1]=>right=>[4,1]=>rig                              =>right=>[9,1]=>right=>[1
0,1]=>down=>[10,2]=>down=>[10,3]=>left=>[9,3
Steps: 37
Press spacebar to exit DFS
```

```
test - Notepad
[7,11]
(2,1)
(7,0) | (9,3)
(1,0,2,2)
(4,0,1,1)
(6,0,2,1)
(1,3,1,2)
(3,3,2,1)
(6,3,1,1)
(2,5,3,2)

Ln 11, Col 1    100%    Windows (CRLF)    UTF-8
```

Figure 17 Test 9



```
Microsoft Visual Studio Debug Console
Currently running Dfs.
failed to get solution
Press spacebar to show DFS for second goal
 Currently running Dfs.
failed to get solution
Press spacebar to exit DFS

C:\IntroToAI\RobotNavigation\bin\Debug\net8.0\RobotNavigation.exe (process 33680) exited with code 0.
Press any key to close this window . . .
```

```
test - Notepad
[5,14]
(0,2)
(4,0) | (9,5)
(2,0,1,3)
(4,0,1,2)
(6,0,1,1)
(2,3,2,1)
(5,3,1,2)
(8,3,2,1)
(3,6,2,3)

Ln 11, Col 1    100%    Windows (CRLF)    UTF-8
```

Figure 18 Test 10

Possible Bugs

1) Inconsistent Grid Dimensions Handling: If the specified grid dimensions (length and width) do not match the actual size of the grid or are not correctly validated, it may result in out-of-bounds errors during node generation or traversal.
2) Inaccurate Wall Node Generation: The code iterates through the specified wall configurations to build wall nodes, but there may be errors with indexing or boundary checks, resulting in inaccurate placement or missing walls.
3) Incomplete Path Generation: If there are barriers or walls that entirely obstruct the path from the agent's starting point to the goal, the algorithm may fail to find a solution or produce an incomplete path.
4) Inefficient Frontier Management: While the frontier management techniques (AddAdjacentNodesToList and AddNodeToFrontier) try to optimise traversal, there may be inefficiencies or corner instances where nodes are wrongly added or removed from the frontier.

Features

1) Modular Design: The code is organized into a single class (Agent), which contains the functionality for a variety of tree-based search techniques. This modular approach encourages code reuse and maintenance.

2) Efficient Frontier Management: The helper methods (AddAdjacentNodesToList and AddNodeToFrontier) optimise frontier management by selectively adding nearby nodes to the frontier depending on particular conditions, such as node eligibility and to avoid redundant exploration.

3) Goal Detection: After completing the primary goal or during the search process, the agent continuously scans the grid to detect the presence of a secondary objective.

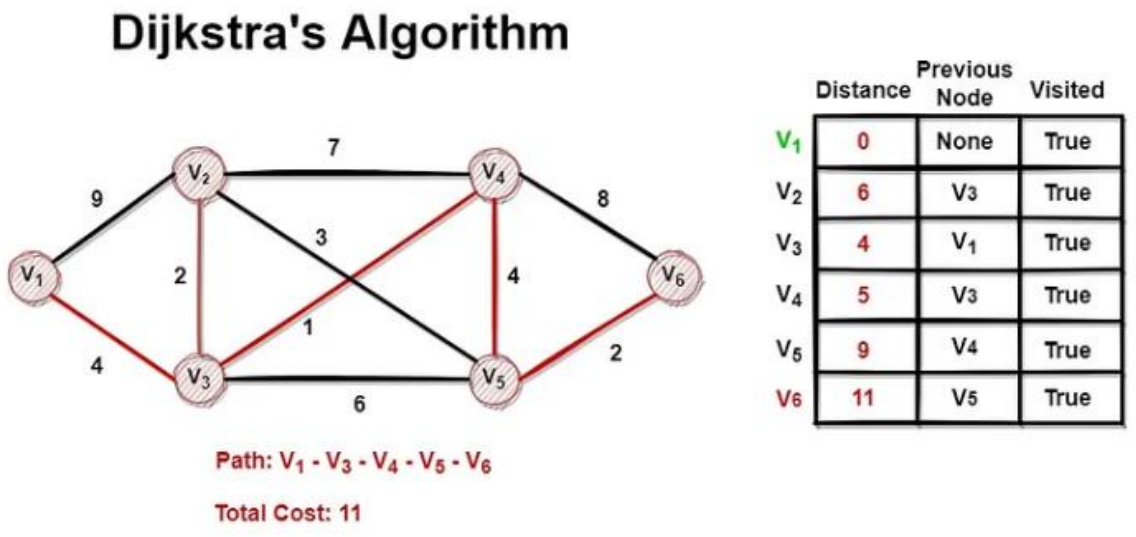
4) Cost-aware Traversal: uniform. Cost Search traverses the grid taking into account the cost associated with each cell. Unlike other search algorithms, which prioritise certain paths or heuristic estimations, UCS examines the grid in a systematic manner by extending nodes with the lowest total cost.

Research

For the custom search I have introduced two search methods namely: Dijkstra Search and Uniform Search. The way they work as follows:

1) DijkstraSearch
   - Initialisation: As with other search algorithms, the approach starts the search process.
   - Main Search Loop: The algorithm iterates while there are nodes on the frontier. In each cycle, the frontier list is sorted according to the cumulative cost of reaching each node. The selected node is removed from the border list and moved to the visited list. If the chosen node is the goal node, the search ends. Otherwise, adjacent nodes are added to the frontier list for future research.
   - Termination: If the goal node is found during the search, the method returns a string indicating that the search has been completed, along with any relevant information. If the search does not find a solution, it will produce a failure message.

An example of this algorithm I used for understanding is shown below:



	Distance	Previous Node	Visited
V_1	0	None	True
V_2	6	V_3	True
V_3	4	V_1	True
V_4	5	V_3	True
V_5	9	V_4	True
V_6	11	V_5	True

Figure 19 Dijkstra's Algorithm

2) Uniform Search

- Uniform Cost Search (UCS) is a fundamental approach for pathfinding and search issues. It traverses the search space by expanding nodes with the lowest path cost, ensuring that it always discovers the least-cost route to the destination.
- In our implementation, we use the Uniform Cost Search algorithm to determine the best path from a start node to a goal node in a grid-based environment. The algorithm works as follows:

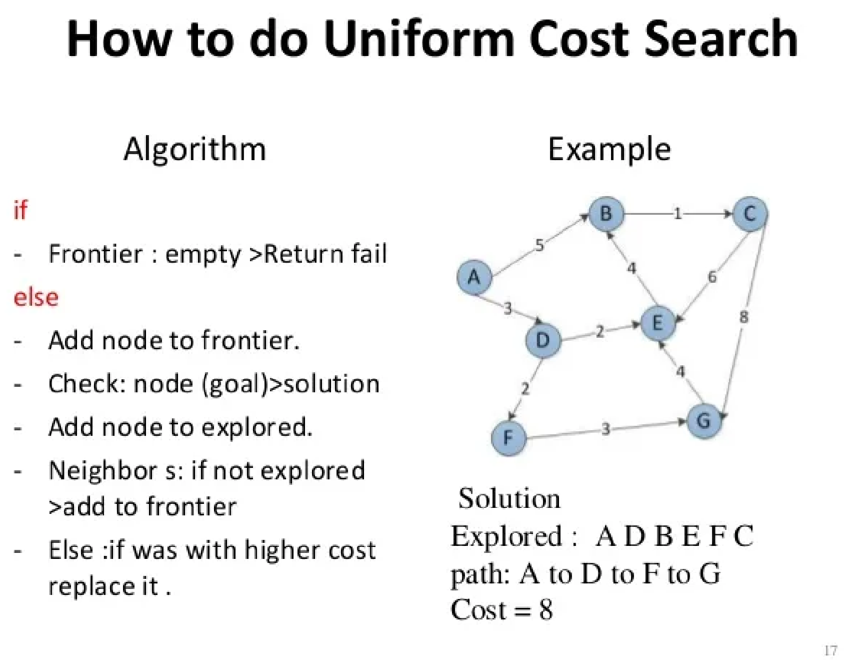
  1) Create an empty frontier list to contain the nodes to be examined.
  2) Add the start node to the frontier for a cost of zero.
  3) However, the boundary is not empty:

     - Choose the node with the lowest cost from the frontier.
     - If the selected node is the goal node, end the search and return the path.
     - Otherwise, add the node's neighbouring nodes to the border, updating their costs.

  4) If the frontier becomes empty before reaching the destination node, the search fails.

Uniform Cost Search performance depends on aspects such as search space size, cost distribution, and efficient data structures for storing frontier and visited nodes.

Similarly to understand the uniform cost search I used the help of some diagrams and websites such as stackoverflow. Below is an example of a diagram that I used to make my code:



How to do Uniform Cost Search

Algorithm

if
- Frontier : empty >Return fail

else
- Add node to frontier.
- Check: node (goal)>solution
- Add node to explored.
- Neighbor s: if not explored >add to frontier
- Else :if was with higher cost replace it .

Example

Solution
Explored : A D B E F C
path: A to D to F to G
Cost = 8

17

Conclusion

In the tree-based search assignment, we used a variety of search algorithms, including Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Best-First Search (GBFS), A* Search, Uniform Cost Search and Dijkstra Search. These algorithms are essential for tackling a variety of search and pathfinding problems.

We discovered many trade-offs in terms of optimality, completeness, and efficiency when implementing and analysing these algorithms. BFS ensures the shortest path but may need a substantial amount of memory, whereas DFS quickly explores the search space but may not find the ideal answer. GBFS and A* Search use heuristic information to direct the search, however this does not guarantee optimal results.

Among the deployed search algorithms, Uniform Cost Search (UCS) emerged as the most optimal. UCS ensures the least-cost path to the goal node by systematically examining nodes with lower path costs.

The optimal search algorithm is determined by the problem's specific needs. However, if we prioritise finding the shortest path while maintaining optimality, A Search* is frequently regarded the best option. A* Search combines the benefits of BFS and heuristic search strategies by utilising a heuristic function to estimate the cost between the current node and the goal node.

References

[1]D. Majetiha, “Uniform Cost Search,” Medium, Oct. 05, 2019. https://medium.com/@dipalimajet/understanding-hintons-capsule-networks-c2b17cd358d7.

[2]A. Soularidis, “An Introduction to Dijkstra’s Algorithm: Theory and Python Implementation,” Medium, Apr. 14, 2022. https://python.plainenglish.io/dijkstras-algorithm-theory-and-python-implementation-c1135402c321

[3]“Tree Based Algorithms | Implementation In Python & R,” Analytics Vidhya, Apr. 12, 2016. https://www.analyticsvidhya.com/blog/2016/04/tree-based-algorithms-complete-tutorial-scratch-in-python/#:~:text=Tree%20based%20algorithms%20are%20considered

[4]“Your One-Stop Solution To Trees in C# | Simplilearn,” Simplilearn.com. https://www.simplilearn.com/tutorials/c-sharp-tutorial/tree-in-c-sharp

[5]“Tree data structure in C#,” Stack Overflow. https://stackoverflow.com/questions/66893/tree-data-structure-in-c-sharp

[6]“What is the best way of using a navigation class in C#,” Stack Overflow. https://stackoverflow.com/questions/11476405/what-is-the-best-way-of-using-a-navigation-class-in-c-sharp (accessed Apr. 19, 2024).

[7]“Uniform-Cost Search (Dijkstra for large Graphs),” GeeksforGeeks, Mar. 25, 2019. https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/