Swinburne University of Technology
Faculty of Science, Engineering, and Technology

COS30019: Introduction to Artificial Intelligence

Assignment 1B: Robot Navigation

Date of report submission	01/05/20
Lab Supervisor	Mahbuba Afrin
Group	10:30am Tuesday

Name	**Student ID**
Jimmy Trac	101624964

This page is mostly blank.

The purpose of this page is to separate the cover page from the report when printed double-sided.

Robot Navigation

Jimmy Trac | 101624964

I. Table of Contents

II. Instructions for Use

The following arguments are used for the program:

```
robot-navigation.exe [filename] [method] <gui/ss> <delay> <true>
```

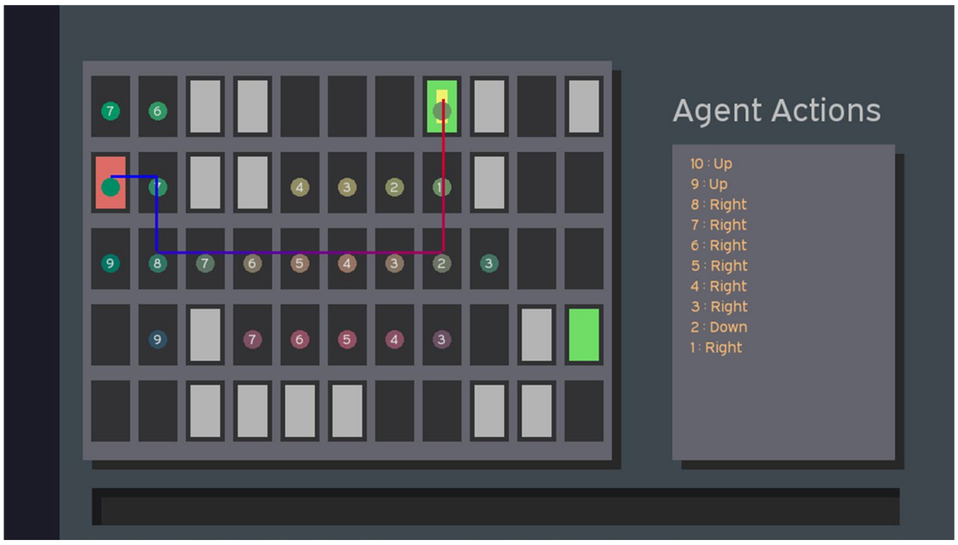
Table II.1 – Arguments and their definitions

Argument	**Definition**	**Example**
Filename	Map filename	RobotNav-test.txt
Method	Agent/Method used in search	bfs, dfs, astar, gbfs…
GUI/SS	Optional: GUI Mode, Screenshot-Output mode, or none for CLI mode	Literally 'gui' or 'ss'
Delay	Optional: Slow down search speed	Non-negative Integers: 0, 5, 10, 20
True	Enable Directional Cost	For now, it only parses 'true'

II. B. GUI Use

The first window is the *Agent Actions* window, which visually represents the map showing the agent starting position, goal positions, and walls. After a short delay, the program will start operating the agent, searching with the desired method until the goal is reached. The program then moves the agent, along the found path and pause at the end.

To close the program, press the Escape (ESC) key.



Agent Actions

```
10 : Up
9 : Up
8 : Right
7 : Right
6 : Right
5 : Right
4 : Right
3 : Right
2 : Down
1 : Right
```

Figure B-1 – Screenshot of the Agent Actions window

The agent actions window shows in colour circles, the nodes that the agent has searched. If the node has an associated cost, it is displayed in the node. Figure B-2 shows the Node Tree window, showing the cost of each path along with the other nodes that the agent was considering in the left.

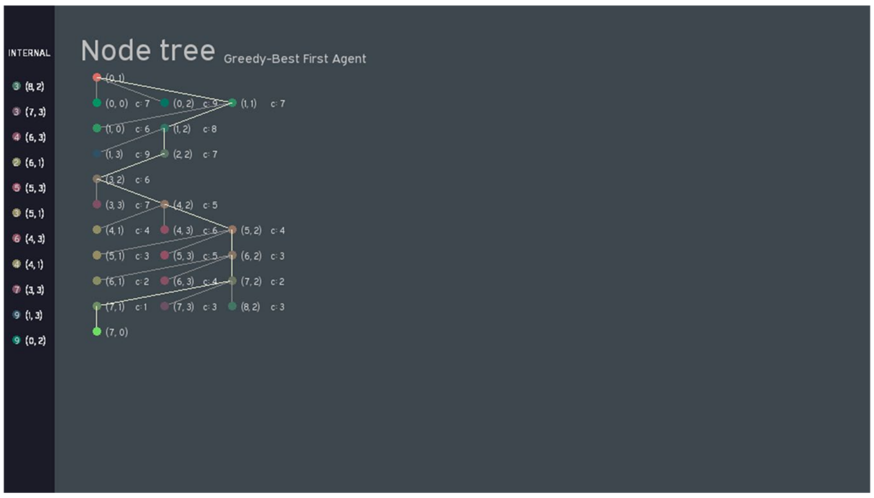


Figure B-2 – Screenshot of the Node Tree window

I. Introduction

I, Robot, Navigate.

Introduction: Introduce the *Robot navigation Problem*, basic graph and tree concepts and other related terminology that may be needed later in your report. (*Hint:* using a glossary)

When we humans navigate, it seems like such a trivial task: point here; point there; avoid all the walls and people in-between. However, as with most human-skills-brought-to-computers, the ability to navigate is a surprisingly difficult and complex task. Whereas human minds make navigation a trivial problem that has had solutions since time immemorial, the use of navigations within robots and robotic systems is a highly valuable endeavour. Whether it be for a home vacuum-bot, an enemy within a game, or for highly complex highly deadly military drones, the ability to navigate terrain poses a highly detailed search problem for computers. After all, when broken down, navigation is the process of getting from Point A to Point B, with numerous states in-between. This is often encapsulated in games, where an enemy agent may try to find the player and navigate to them. In such cases, the robot navigation problem is simplified, as the environment is deterministic and well-known.

Problems that require searching can be broken down into a series of decisions made by an agent or player. For example, in the case of chess, a player has numerous decisions for their next move, of which, each of those decisions are then followed by a subsequent one by the second player. Then laid out, these possible future states form a *search tree* such as figure 1.1.

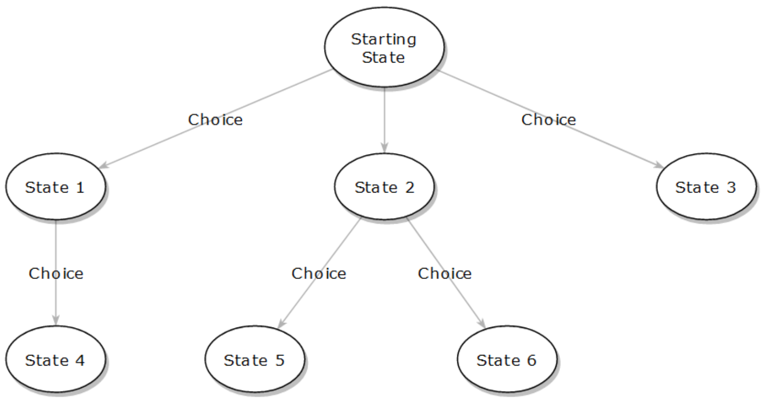


Figure 1.1. – An example search tree composed of states

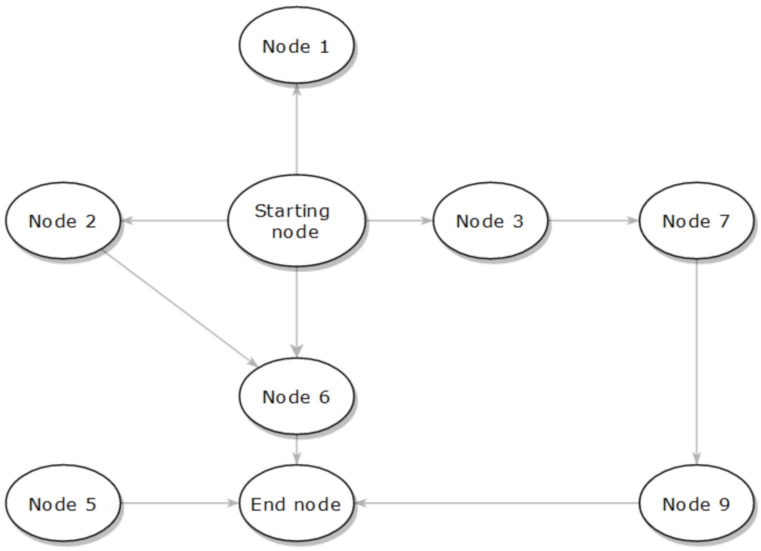


Figure 1.2. – An example graph, identical to Figure 1.1, composed of states.

In contrast, graphs are loosely connected nodes without the hierarchal structure seen in search trees. They can have loops and do not have a root node. They are more of a network, and programmatically, they can be traversed by simply not keeping a record of visited nodes in a problem space. This is useful for searching efficiently without memory use but oftentimes unproductive as uninformed agents such as Breadth-First Search may try to search all of the current level before passing onto the next, such as all the squares surrounding an immediate position, without realising that there are only four, and the agent as done so for the last hundred or so searches. Bringing it all together, the robot navigation encompasses a significant amount of tree search and the complexity of the problem is such that the method in which we traverse the tree becomes a significant factor for timely completion. This brings us into search algorithms:

1.1. The Robot Navigation Problem

In this report we will tackle the *Robot Motion Planning* or *Robot Navigation* problem. The navigation problem deals with the ability to get from a starting node to the goal node whilst avoiding obstacles. To more rigorously define the problem, we lay out a set of rules for the environment and the agent itself.

Simply put, the environment is:

- A 2-dimensional grid of size N x M,
- Static and non-changing,
- Composed of tiles in each grid square, where a tile can consist of the following:
  - Empty Space;
  - A wall;
- Furthermore, there additional types of tile:
  - The goal tile(s), where there is a minimum of one;
  - The starting tile, where the agent is initially placed; and
- There is only one agent.

The rules for the agent are that it:

- Only perceives the world through pre-defined *percepts*;
- Can only move up, down, left, or right;
  - It should be noted that the agent should ideally not move when searching;

    - Cannot move diagonally;
- Cannot move into or over walls;
- Cannot move beyond the map boundaries;

The goal of the navigation problem is to move the agent to the goal position in the least amount of actions possible, in the least amount of time, and using the least amount of memory. An example map of the Robot Navigation problem can be seen in Figure 1.1.1, showing a grid with the Agent along with Walls and the goal position.

Table 1.1.1. – An example Robot Navigation Problem

		Wall		
		Wall	Goal	
Starting Tile		Wall	Wall	
		Wall		

1.1.1. PEAS of the Task Environment

To more formally describe the agent and environment, we employ the PEAS metric. PEAS refer to the Performance Metric, Environment, Actuators, and Sensors given within an environment. (Novig & Russell, 2009, p. 40). Given the environment of the navigation problem, we can evaluate the PEAS metric to the following in table 1.1.2.

Table 1.1.2. PEAS for the navigation problem

Performance Metric	**Environment**	**Actuators**	**Sensors**
Number of Steps Taken + Time taken to reach goal. Goal would be to Minimise.	2D Grid, Walls, Map Border.	Movement: Up, Down, Left, Right.	Knowledge of Map, Location of starting and end nodes, current location.

1.1.2. Properties of the Task Environment

In addition, we can break down the task environment into more formal parameters given the known rules. Specifically, we refer to the following factors, whether the environment:

- Is Fully or Partially Observable;
- Has multiple or a single Agents;
- Is Deterministic or Stochastic;
- Is Episodic or Sequential;
- Is Static or Dynamic; and
- Is Discrete or Continuous.

(Novig & Russell, 2009, p. 45)

Given the environment of the navigation problem, we can describe it as laid out in Table 1.1.3.

Table 1.1.3. – The Task Environment broken down into its properties as described by Novig and Russell.

Observable?	Agents?	Deterministic?	Episodic?	Static?	Discrete?
Fully	Single	Deterministic	Sequential	Static	Discrete

Observable: The task environment is fully observable as the agent's sensors will provide the entire map to the agent. This includes the locations of the starting and goal positions, along with the agent's current position.

Agents: There is only one agent in the map, hence, it is a singular-agent problem.

Deterministic: A deterministic environment is completely determined by the actions of the agent. We know that there are no uncertainties within the environment.

Episodic: If we state that each movement of the agent is an episode, we can surmise that any agent the action takes will be determined by the previous movement, as the agent changes positions.

Static: An environment is static if it changes as the agent is deliberating. In the case of the environment, as there are no additional factors or forces acting on the agent, it is completely static.

Discrete: Referring to the continuity of the *time* or the percepts of the agent, the environment of the navigation problem can be described as discrete. The time within the world is handled in discrete steps with no updates in-between.

II. Search Algorithms

Getting from Point A to Point B is B Minus A.

When we lose items that we need, we look for them. The act of searching is a simple yet highly valuable human task; searching requires co-ordination, the ability to recognise objects, past experience ('where do I typically leave my keys?'), heuristics ('a laptop cannot hide under a pile of paper'), and much, much more. When taken to its fundamentals, searching is the ability to locate a goal state and then perform a resolute action. In the case of lost keys, it would be to pick them up and swear that you did not place them there. This is an important distinguishing factor – *Searching for Keys* and subsequently *Going to Pick Up the Keys* are two separate processes and should not be confused with one another. When brought to Robot Navigation, the ability to *Find* the goal node is one process, whereas *Getting to the Goal Node* is another; this is this, that is that.

When we do search for perpetually-lost items, we are often unwilling to spend an almost infinite amount of time doing so – there are better things to do with our resources. The case is the same with search algorithms (a programmer term for 'search techniques'), as we want to find the goal state, and subsequently get to it, in the least amount of time, using the least amount of space (memory). Here we can talk in detail about the mathematical description of algorithms and their efficiency and/or effectiveness. If your friend finds their keys in five minutes, yet you cannot; it does not mean that your friend will always be better at finding keys than you. In the same vein, if one algorithm works better on one computer, it may not necessarily translate to another. Maybe your friend had their glasses on, or you have completely missed a highly obvious location. There are a multitude of factors that make benchmarking – running a program on a computer and to measure the time taken or some other score – a relatively bad idea for comparing algorithms. Here, we lightly touch on the concept of Big-O notation, or O() Notation, or space-slash-time complexity.

2.1. Space and Time Complexity: Big-O Notation

Also known as Asymptotic Analysis/Notation, big-o notation is the idea that when the number of times an algorithm's input is brought to some unknown number near infinity, the time taken will be function of that number, *n*.

For example, if an algorithm takes one second to sort two elements, and five seconds to sort 10, we can roughly say that the algorithm has a linear complexity – O(n). This contrasts with another sorting algorithm, which might have an O(n^2) complexity, taking four seconds to sort two, but 100 to sort 10.

For a more detailed discussion and analysis on Big-O notation, the reader is first directed to the Wikipedia page and then to Artificial Intelligence: A Modern Approach, Appendix A. For brevity, common orders and their orders are given in table 2.1.1.

Table 2.1.1. – A short table of common O-Notations, ordered from least complex (top) to most complex (bottom)

Notation	**Name**
O(1)	Constant
O($\log n$)	Logarithmic
O(n)	Linear
O(n^2)	Quadratic
O(n^c)	Polynomial
O(c^n)	Exponential
O($n!$)	Factorial
O(n^n)	A lost cause

2.2. Informed and Uninformed Algorithms

When faced with a problem, there are one of two possible cases that an individual may be in: they understand the problem, or they do not. The separation between informed and uninformed algorithms lies in whether there is additional guidance provided about a problem. For example, given a pair of lost keys, if we do not have any understanding of the nature of searching for keys, we can simply blindly search following a strategy to keep ourselves on track. On the other hand, if we know that we tend to leave keys near a certain location, we can *inform* ourselves on our search and potentially search in a more efficient manner.

When we describe informed and uninformed algorithms, we refer to the method in which a search tree can be traversed. Using the example search tree in figure 2.1, we aim to lightly explore concepts related to search algorithms. For additional information, the reader is directed to Chapter 2 of Novig & Russell's AI: A Modern Approach.

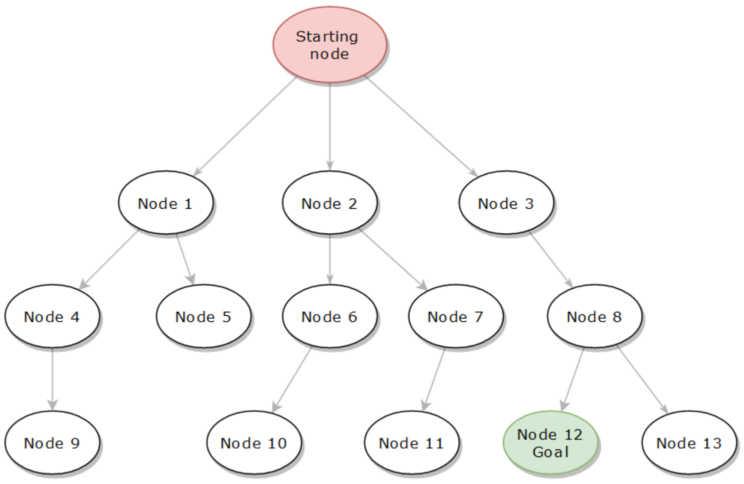


Figure 2.1. – An example search tree with a starting node and a Goal Node

Say, for example, we have no idea where the goal node is and can only confirm that a node is the goal by stumbling over it. A blind stumble, in other words. We may say that a simple way to search is left to right, and therefore, our search path resembles Figure 2.2. This form of left-right traverse is known as a Breadth-First Search (BFS). An uninformed algorithm. Uninformed algorithms by their nature are blind stumbles due to the absence of guidance within a problem, and therefore, can be mightily inefficient, yet, despite the drawbacks, if there is a solution, it will be found.

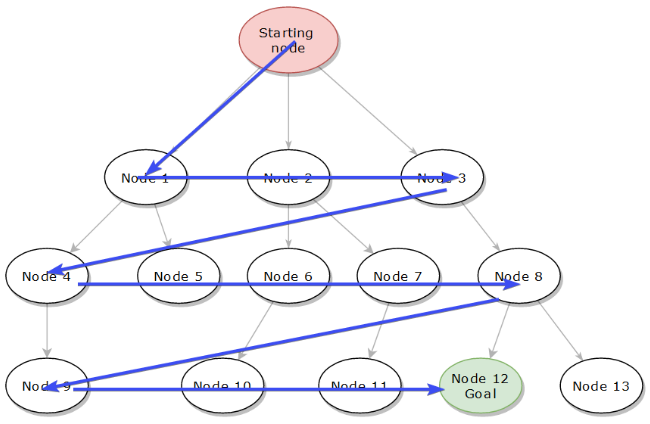


Figure 2.2. – A Breadth First Search on the example tree

If we perchance are given slightly guidance in the form of some cost, we can search our tree more efficiently. Figure 2.3 shows the same tree, however, there are slight costs associated with travelling to the next node. Perhaps the cost gives a rough estimate to the goal, perhaps it is related to the distances between nodes. The algorithm needs not know; the algorithm needs not care; only for the numbers, does the algorithm move. If we traverse the tree taking a lowest-cost approach, we search in the manner given in figure 2.4.

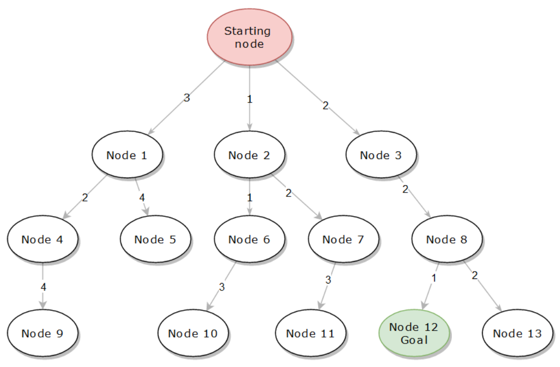


Figure 2.3. – The search tree as in figure 2.1 with additional costs

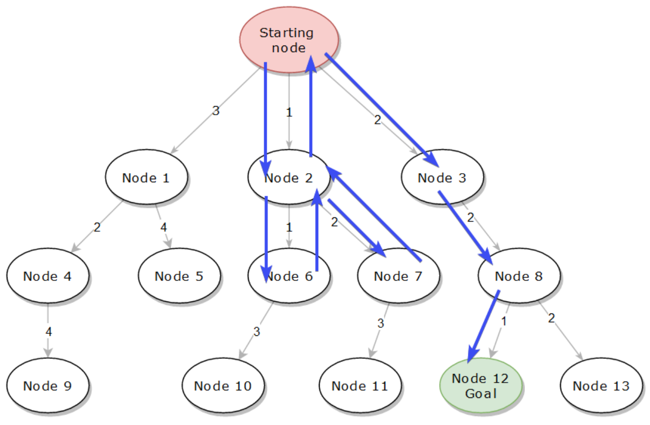


Figure 2.4. – A search on the cost-added graph, taking only the lowest cost at each node. A Greedy-Best First Search.

Compared to our initial meander without any guidance, an informed search is capable of much efficient and faster searches as they are often more directed. This direction, however, is highly dependent on the costs we give to each node – the cost function. The cost of any node is the manner in which informed algorithms are given their 'smarter' search patterns, however, as we will see, the differentiating factor in different informed search algorithms is how their cost functions are structured or which guide – or heuristic – they utilise.

2.3. Search Algorithms

The following search algorithms will be explored in this report, broken down into two groups:

Uninformed Algorithms

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)
3. Uniform Cost Search (UCS)*
4. Iterative-Deepening Depth-First Search (IDDFS)*

Informed Algorithms

5. Greedy-Best First Search (GBFS)
6. A-Star Search (A-Star)
7. Dijkstra's Algorithm (DJA)*

*UCS, IDDFS is the CUS1 requirement whereby DJA is CUS2.

A quick explanation of each is as follows.

Breath-First Search: On a search tree, it is to search left-to-right, once the rightmost node is reached, move down one level. On the application level, a queue is used to store nodes to search. For any one node, its child nodes are enqueued, after which, the next node in the queue is searched.

Depth-First Search: On as search tree, it is to search down-then-right. Once the bottom is reached, return to the previous node until another child node is found, and proceed down that branch. On the application level, this requires a Stack in which child nodes are pushed to. The next node is popped from said stack.

Uniform Cost Search: The cost of each node is the number of steps to reach it. The next node is the lowest cost node, using a priority queue. Of note, on an explored node, its children's is evaluated and stored in queue.

Iterative-Deepening Depth-First Search: Like Depth-First Search, however, there is a depth limit which is increased every time the search completes. This does require repeating all the previous searches.

Greedy-Best First Search: An informed search where a Cost Function is used to evaluate the validity of each node. The node is then added to its priority queue where the next node is the least-cost.

A-Star Search: On top of the Cost Function, a Heuristic or guided function is added to evaluate the cost to move to a certain node. In other words, the cost of the node is the sum of 'the cost to get to the node, and the cost to get to the goal'.

Dijkstra's Algorithm: The predecessor of A-Star, the cost function is a heuristic function that determines the cost to the node itself.

2.4. Experimental Method and Results

Though it should be noted that benchmarking that benchmarking on any one computer is a poor comparison, the aim of this report is to explore the efficiency and effectiveness of the search algorithms to each other. Comparisons between the search algorithms will be based on the number of nodes expanded by any agent and the number of steps in the final path. The following batch script (cut down) was used to run the test suite given in Code 2.4.1.

Code 2.4.1. – Batch script used to run the test suite

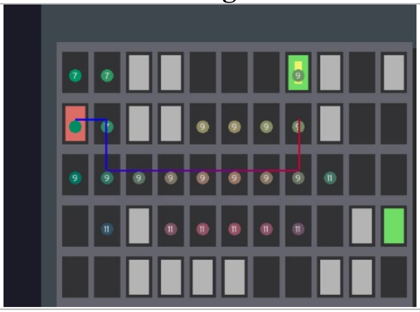
```
set _agents=bfs,dfs,gbfs,astar,ucs,iddfs,dja
set  _maps=RobotNav-test.txt,robot-nav-map2.txt,robot-nav-map3.txt,robot-nav-map4.txt,robot-nav-map5.txt
set _APPNAME=robot-nagivation.exe

for %%G in (%_agents%) do (
    for %%H in (%_maps%) do (
            %_APPNAME% %%H %%G ss 0 true >> %_logfile%
            %_APPNAME% %%H %%G ss 0 >> %_logfile%
    )
)
```

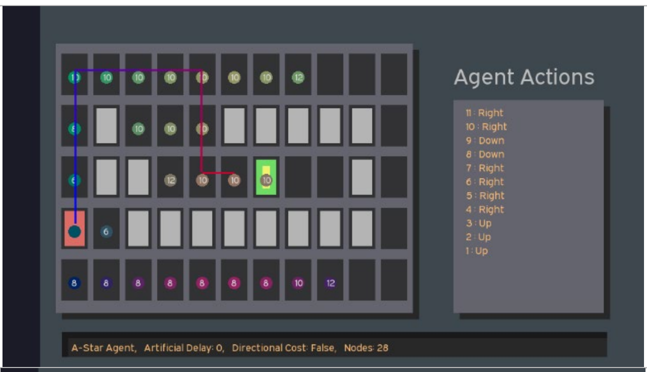
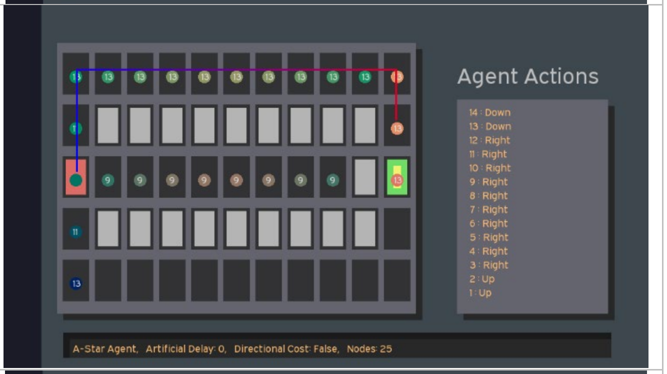
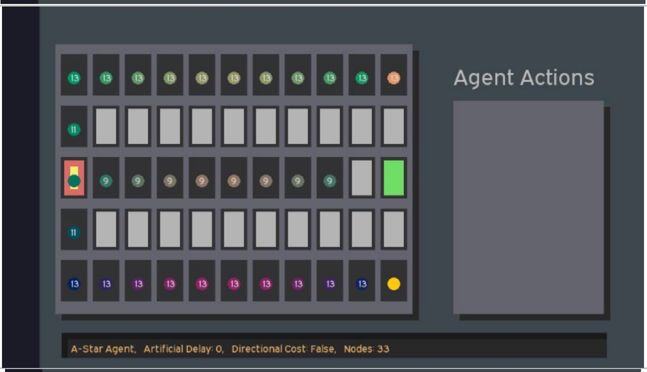
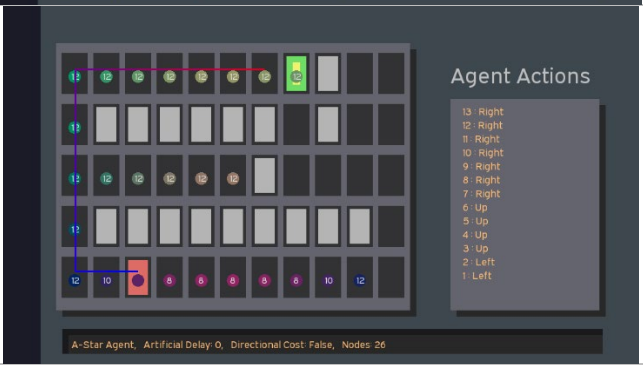
The data measured includes:

- The number of nodes in memory (space complexity)
    - This is measured as the count of the internal queue/stack/list of the agent
- The number of nodes along the search path (idealness of path)
    - This is measured as the count of the number of steps taken by the agent
- Screenshots of the path and node tree

What the data does not include, is **time complexity**. The amount of time taken to run a test was not measured as the outputs relied on the standard output requirement of the task. Hence, no additional information was provided. Table 2.4.1. shows all the map layouts as given by the A-Star agent.

Table 2.4.1. – All 5 Maps traversed by the A-Star Agent

Map	Image
Robot-Nav-Test	

Map 2	 Agent Actions 11 : Right 10 : Right 9 : Down 8 : Down 7 : Right 6 : Right 5 : Right 4 : Right 3 : Up 2 : Up 1 : Up A-Star Agent, Artificial Delay: 0, Directional Cost: False, Nodes: 28
Map 3	 Agent Actions 14 : Down 13 : Down 12 : Right 11 : Right 10 : Right 9 : Right 8 : Right 7 : Right 6 : Right 5 : Right 4 : Right 3 : Right 2 : Up 1 : Up A-Star Agent, Artificial Delay: 0, Directional Cost: False, Nodes: 25
Map 4	 Agent Actions A-Star Agent, Artificial Delay: 0, Directional Cost: False, Nodes: 33
Map 5	 Agent Actions 13 : Right 12 : Right 11 : Right 10 : Right 9 : Right 8 : Right 7 : Right 6 : Up 5 : Up 4 : Up 3 : Up 2 : Left 1 : Left A-Star Agent, Artificial Delay: 0, Directional Cost: False, Nodes: 26

2.5. Discussion of Search Algorithms

Running the test suite, we resolve 70 tests across 5 different maps, with 7 agents, of which, an additional test includes directional movement cost. The data is as follows in Table 2.5.1. The raw data is archived and should be accessible via the GitHub Repository. This section will be broken down into a discussion of the respective agents, along with observations in the final section.

Table 2.5.1. – Space and route complexity of the various algorithms

Space Complexity

Agent Type			Map				No Solution			
	Map 1 D	Map 1	Map 2 D	Map 2	Map 3 D	Map 3	Map 4 D	Map 4	Map 5 D	Map 5
BFS	34	34	31	31	36	36	33	33	36	36
DFS	28	28	15	15	15	15	34	34	14	14
GBFS	28	24	31	33	24	25	33	33	19	19
A*	31	24	25	28	36	25	33	33	28	26
UCS	39	45	38	32	35	30	34	34	38	37
IDDFS	24	24	14	14	14	14	33	33	13	13
DJA	30	31	23	23	34	32	33	33	24	25

Route Complexity

Agent Type			Map				No Solution			
	Map 1 D	Map 1	Map 2 D	Map 2	Map 3 D	Map 3	Map 4 D	Map 4	Map 5 D	Map 5
BFS	10	10	11	11	14	14			13	13
DFS	20	20	13	13	14	14			13	13
GBFS	10	10	26	26	14	14			15	15
A*	10	10	11	11	14	14			13	13
UCS	12	10	11	11	14	14			14	15
IDDFS	14	14	13	13	14	14			13	13
DJA	10	10	11	11	14	14			13	13

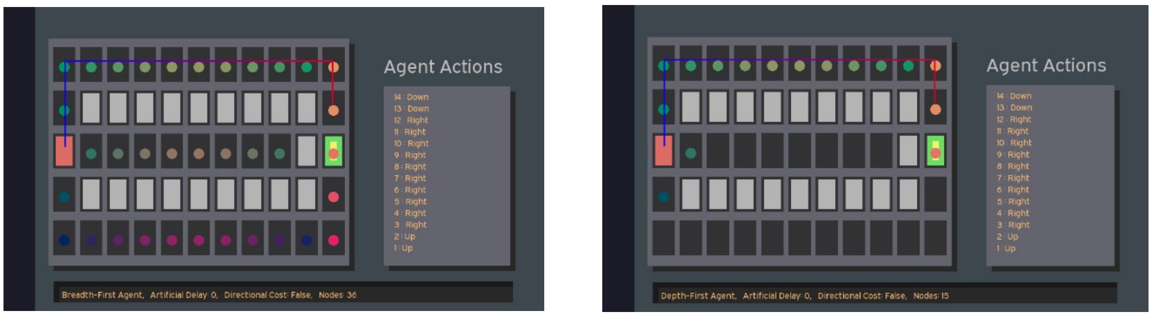
2.5.1. Uninformed BFS, DFS, UCS, and IDDFS

The experiment revealed crucial differences between the various algorithms. Notably, the difference between blind and non-blind *uninformed search algorithms.* In particular, Depth-First Search (DFS) and Breadth-First Search (BFS) do not rely on cost in any way to determine search pattern, and therefore show no difference between regular or directional tests. This is in contrast with Uniform Cost Search (UCS), which does incorporate a cost function, albeit an uninformed one. This may come down to the decision of the developer to enable or disable the effects of directional costs on UCS.

2.5.1.1. Breadth-First Search and Depth-First Search

The quintessential uninformed algorithm, BFS provides the most ideal route provided it exists. This can be seen experimentally in Table 2.5.1, where BFS is able to provide the shortest route for all maps. Unfortunately, this is coupled with rather high space complexity compared to all other algorithms, falling behind Uniform Cost Search in Map 1.

Compared to Breadth-First Search, DFS provides a very low-memory strategy for getting to root nodes provided they are very deep in the tree. In this case relating to robot navigation, oftentimes the goal is quite deep and favourable for DFS. This shows in the various cases for space complexity, where the goal was reached quite quickly in Maps 2, 3, and 5. In map 2 shown in figure 2.5.1.1 in shows how the depth-searching and the direction priority (up, then left, then right) aids in the quick searching of certain goals. The figure shows BFS performing a search on almost the entire search space whereas DFS skips the middle trap entirely. Of course, this is not without drawbacks, as oftentimes the path provided by DFS is unideal.



```
14 : Down
13 : Down
12 : Right
11 : Right
10 : Right
9 : Right
8 : Right
7 : Right
6 : Right
5 : Right
4 : Right
3 : Right
2 : Up
1 : Up
Breadth-First Agent,  Artificial Delay: 0,  Directional Cost: False,  Nodes: 36
```

```
14 : Down
13 : Down
12 : Right
11 : Right
10 : Right
9 : Right
8 : Right
7 : Right
6 : Right
5 : Right
4 : Right
3 : Right
2 : Up
1 : Up
Depth-First Agent,  Artificial Delay: 0,  Directional Cost: False,  Nodes: 15
```

Figure 2.5.1.1 – Comparing BFS and DFS in Map 2

In a relatively straightforward map such as the provided one, DFS performed poorest taking 20 steps to reach the goal compared to the 10-12 of every other algorithm. Hence, the performance of DFS is highly situation depending on the layout of the goal node within the search tree and the order of searching.

2.5.1.2. Uniform Cost Search and IDDFS

The custom uninformed algorithms chosen for the assignment were Uniform Cost Search and Iterative-Depth First Search. The reason for choosing both was to compare the space complexity requirements, as they both have unique methods of memory management. Though time complexity was not measured as a part of the experiment, empirically, IDDFS took significantly longer to run any one map compared to UCS due to the re-pathing nature of the algorithm. As it had to re-do all the previous work, it took significantly longer than straightforward UCS.

Uniform Cost Search performs the worst in terms of space complexity due to its evaluation requirement. UCS specifically evaluates all neighbouring nodes in its frontier expansion to evaluate the cost and therefore, takes up significantly more memory than all the other algorithms, at 45 nodes for Map 1. This contrasts with the space complexity for IDDFS, which is the lowest of all algorithms even compared to regular DFS. Specifically, the depth-limiting factor improves the performance of IDDFS in open maps where each branch has significant depth, as on map 1 / the test map, requiring the expansion of only 24 instead of 28 nodes, the lowest of all the algorithms.

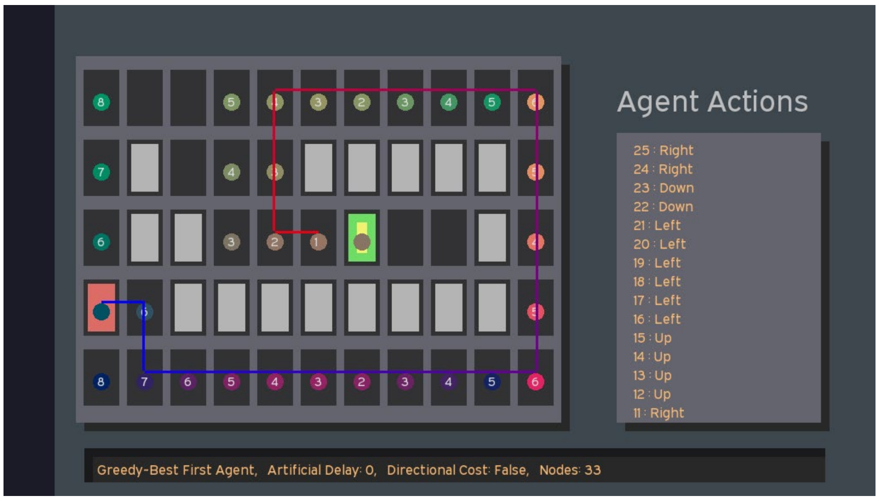
Ultimately IDDFS versus UCS shows the exchange between space, time, and route complexity. Where IDDFS will always use less memory than DFS and "just" enough to reach a certain depth, IDDFS has the highest time requirement, empirically. Compared to UCS, a relatively quick but memory-intensive algorithm, they represent two distinct methods in which to apply to specific situations. In the case of IDDFS, it is ideal where time is not a factor, but space is, and vice-versa for UCS. However, this is only the case for uninformed algorithms. With appropriate heuristics, informed algorithms are expected to perform better.

2.5.2. Informed GBFS, A-Star, Dijkstra's Algorithm

Informed searches are characterised by their additional heuristic functions, providing more efficient searching of any state-space. The following section details the performance of GBFS, A-Star, and Dijkstra's Algorithm in the robot navigation search space.

Using the Manhattan distance between the node and the first goal, GBFS presents a significant advantage over BFS and DFS in space complexity, especially in open maps such as Map 1 and 5 in figure 2.5.1. GBFS does show issues relating to certain maps, such as Map 2 as designed to exploit the cost function provided to GBFS, as seen in figure 2.5.1.2, showing its flaws on Map 2 and 5. The observed behaviour is due to the cost function being only the distance to the goal, ignoring walls. This contrasts with Dijkstra's Algorithm, which uses *the cost to get to the node* instead. The combination of a cost function and GBFS's heuristic leads to the A-Star algorithm.

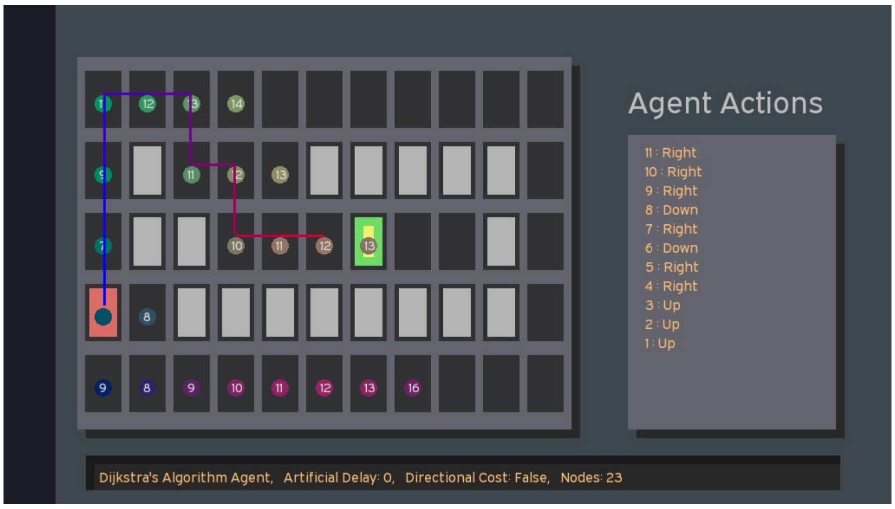
In space complexity, the various informed algorithms are matched only by each other, and are often significantly better when compared to uninformed algorithms. This is especially true in Map 5, where all the informed algorithms perform significantly better due to the forking design of the map compared to BFS and UCS. This leads to overall better memory performance at the cost of computational power and admissible heuristics.



25 : Right
24 : Right
23 : Down
22 : Down
21 : Left
20 : Left
19 : Left
18 : Left
17 : Left
16 : Left
15 : Up
14 : Up
13 : Up
12 : Up
11 : Right

Greedy-Best First Agent, Artificial Delay: 0, Directional Cost: False, Nodes: 33

Figure 2.5.2.1 – Greedy Best First Search taking a very flawed route



11 : Right
10 : Right
9 : Right
8 : Down
7 : Right
6 : Down
5 : Right
4 : Right
3 : Up
2 : Up
1 : Up

Dijkstra's Algorithm Agent, Artificial Delay: 0, Directional Cost: False, Nodes: 23

Figure 2.5.2.2 – Dijkstra's Algorithm finding the ideal path

Ultimately the informed algorithms perform better than the uninformed algorithms, except DFS on favourable maps, due to their better space complexity and often optimal paths. Specifically, Dijkstra's Algorithm and A-Star show in experiment that there a solution exists, they will provide the shortest and most efficient path to the goal state.

III. Implementation

Design and Development of the program

3.1. Top-Level design

Designed after the Model-View-Controller Paradigm, the navigation program is first and foremost designed with the GUI implementation in mind.

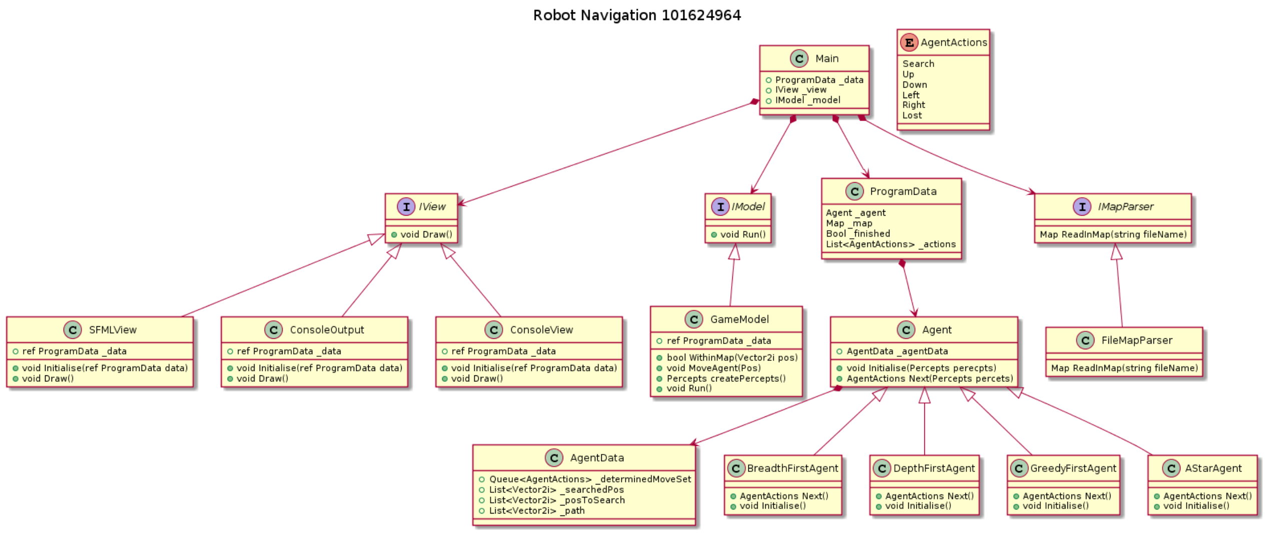


Figure 3.1. – A high level overview of the navigation program

The program is modelled after three main components, the:

- Model: Contains all the world logic and "runs" the simulation
- View: Encapsulates all the GUI functions and does not manipulate any data
- Controller: Passes the message objects between the Model and View and initialises both.

Hence, the program would be able to maintain both a GUI output along with various other output views, such as standard output or even console output based on the supplied view by the Controller alone. This enables a significant amount of flexibility in designing the GUI within the program as it is not directly tied to any simulation or logic.

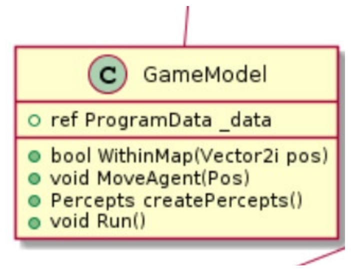


Figure 3.2 – the Game Model

The model itself contains all the logic of the world. Specifically, it has the member functions given in figure 3.2, allowing itself to run the simulation on a step-by-step basis and exposing logic for agents to use – in this case, whether it is within the Map or movement logic. Furthermore, it is the model that creates the percepts from the primary data object, allowing for strict data control of the agent in highly sterile environments.

The agent itself is a virtual class that relies on inheritance to create informed and uninformed agents, as given in figure 3.3.

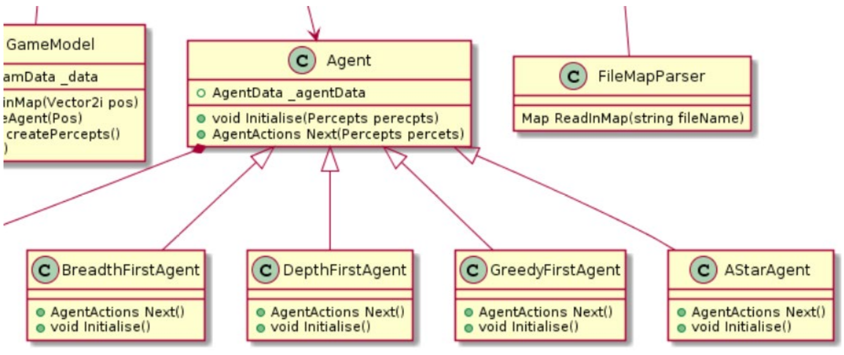


Figure 3.2 – The agent inheritance structure

The agents themselves strictly only have one function: Next. The next function returns the agent's decisions, and if the agent is still searching, it returns a "searching" action. The agent itself therefore manages its own state and exposes its internal data as an object for the program data to access. As the agent itself is not stored within the model, but within the primary data object, the rest of the program has access to the agent's internal information and state. This is primarily used to generate the search patterns and associate the costs as seen in the primary GUI views.

A key difference between the navigation program and the standard algorithms is the loop. Specifically, the agents themselves do not loop, but run only one stage of their search per call, allowing for real-time viewing of their search patterns and movement. This is not true of normal search algorithms, which will lock themselves into a while loop until the search is complete. This significantly slows down the searching algorithms, even at optimal speed.

3.2. Features/Bugs/Missing

This will be a quick summary of the features, the bugs, and any missing components of the robot navigation program. A more thorough explanation of the GUI implementation can be found in Section 4: Research.

3.2.1. Features

- GUI Map display, node search, and path
- GUI Node Search Tree
- Command Line Interface
    - Standard Output
    - Console Output (Dated)
    - GUI Output
- GUI Output allows artificial slowing for searches
- Directional Cost
- Has a "Test Mode" which includes screenshot and data writing, for use with the associated Test Suite batch file

3.2.2. Bugs

- The delay at the start is not a bug! It is intentional to allow the user to move windows before the program starts!
- The program does not perform extensive checking for CLI inputs
- The program will not be able to provide specifics when loading in a faulty map, it just returns the error and ends the round
- The program relies extensively on robustly built agents, and the agents themselves can become quite complex due to the GUI requirements.

- The node tree can be quite complex and convoluted in certain maps, this is not fixable without an overhaul of the node tree code
- The View code is almost completely procedural. This was done for timeliness as making a complete composite UI system would take too much time.

3.2.3. Missing

- Does not provide any information about time complexity nor number of steps for a given search algorithm
- All pre-provided agents are designed to track the first goal node down, and not any other. This can be remedied by modifying the agents themselves for further research
- The agents are quite complex and are-far removed from their pure algorithmic counterparts, largely due to the GUI requirement.
- The agents provide very little structure for how they should be run, which was originally designed for flexibility but is now completely dependent on copying and pasting existing agent code.

IV. Research

Additional Extensions

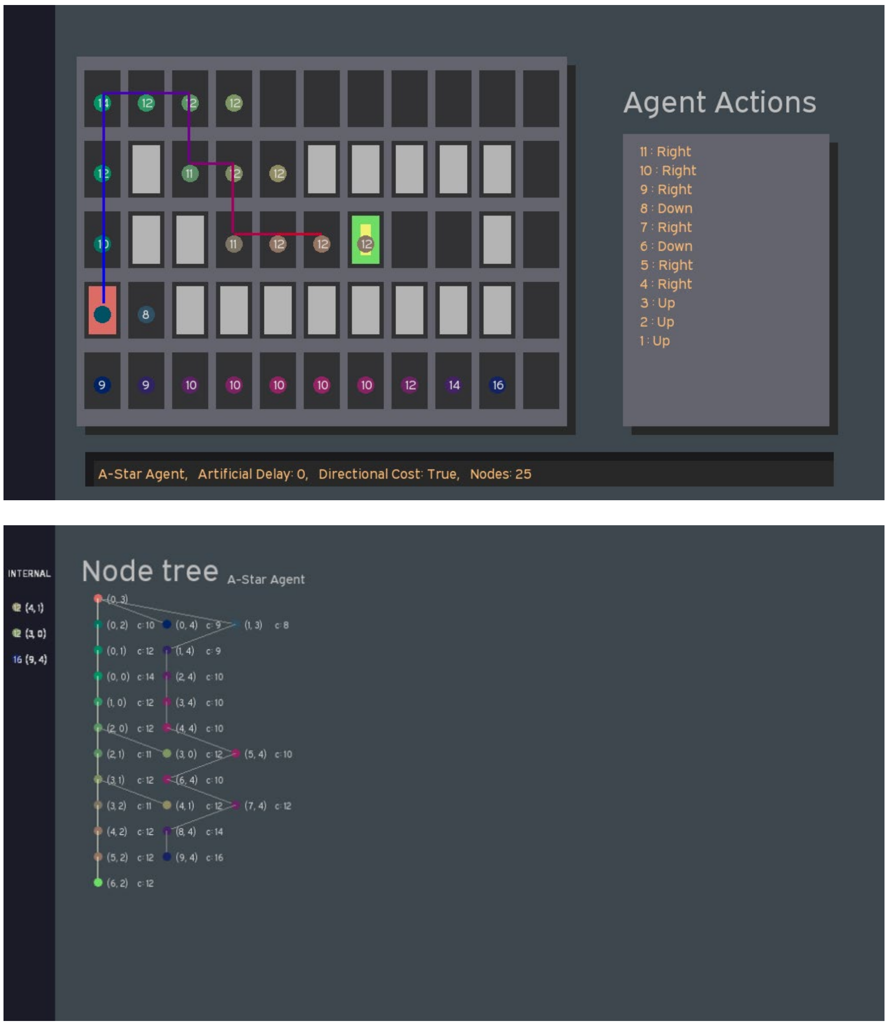
As described throughout the report, various research intiatives were undetaken and they will be described in detail here.

From the assignment, the following research initiatives were taken:

- GUI Implementation
- Directional Costs and Comprehensive testing

4.1. GUI Implementation

The robot navigation program accepts a minimum of two arguments as the standard command-line interface (CLI) output. Beyond this, the program is designed for extensive GUI use by specifying a third argument 'gui'.


Agent Actions

```
11 : Right
10 : Right
9 : Right
8 : Down
7 : Right
6 : Down
5 : Right
4 : Right
3 : Up
2 : Up
1 : Up
```

```
A-Star Agent,  Artificial Delay: 0,  Directional Cost: True,  Nodes: 25
```

Figure 4.1. – The two GUI windows showing an A-Star agent search

The primary features of the navigation program are:

- A two-window GUI system to display the movement of the agent along with a representation of the node tree search-space as seen in figure 4.1.
- The agent's search pattern is represented in real-time, along with the node tree. This shows the evolution of the search in both tree and graph form for comparison.

The program was designed with the following design goals in mind:

- To be as simple as possible, without highly complex GUI elements
- To be easy on the eyes (focus on colour use, especially for accessibility)
- To be "screen-shottable", that is, all information that the user needs is displayed at any one moment in time. This can be seen in features such as the coloured movement path or the description of the agent along with any additional information in the agent action's debug box.
- To have all information displayed at any one time
    - This means that switching between tabs is not useful, as we want to be able to see both the evolving node tree and the agent search at the same time, hence, two windows.
- To expose all information used by the agent, so that the user and the agent are 'on the same page'
    - This was implemented in the form of the internal information stack on the left of the Node Tree view.

The Agent Actions Window has the following features:

- Contains a graphical representation of the map
- Searched nodes are highlighted with a coloured circle corresponding to the position, this is also the same colour as the node view.
- The cost is displayed in the centre of the circle
- A path is drawn to show the movement of the agent, from Blue (cold) to Red (hot) to show the movement of the agent in still captures.
- The actions and all additional information is displayed in the single

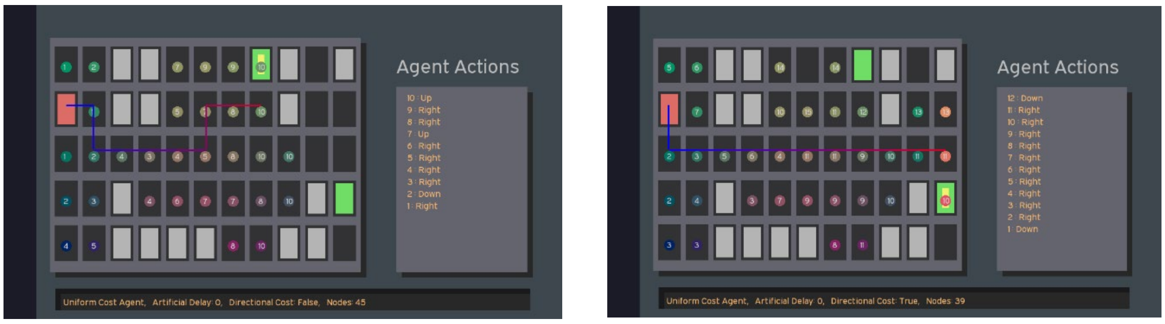
The Node Tree View has the following features:

- The tree evolves dynamically! This is apparent in Depth-First Search and similar methods, which shows the backtracking.
- The tree shows the current nodes that are being searched in yellow/orange in real-time.
- Nodes are coloured based on their position, for example, a position in the left will always show as green-tinted and as the node moves south, becomes increasingly pink. This colour corresponds with the same position as the agent actions screen.
- The cost and direct position of each node is shown
- The tree is an accurate representation of the search space, where every layer is indeed the "generation" of that search.
- The final path is highlighted
- The left bar of the node tree is the internal stack/queue/priority queue/list that the agent uses. This is most useful in A-Star or Greedy searches which show the other nodes the agent can consider when choosing the least-cost node.
- The node tree can scroll up/down to accommodate deep trees. Easing was introduced to the scrolling function to make it less jarring to use. As described by the author, extended periods of *bad scrolling functions really sucked.*

4.2. Directional Costs and Extensive Testing

Directional costs were implemented as a part of the cost function in the base agent, allowing for a simple program toggle. The results were discussed extensively in section 2, however, it will be touched on slightly here.

As table 2.5.1. suggests, the searching cost increase due to direction has a significant impact on the path the certain algorithms take. Uniform Cost Search is the most heavily affected, as it ventures to the second goal instead of the first. This is due to the uninformed nature of UCS, as it concerned itself with the node that is the closest to the goal, cost wise, instead of specifically goal 1 in which the other algorithms focus on. This is seen in figure 4.2.1, showing that without the directional cost, it is easier to get to the first goal whereas with directional cost, it is easier to get to the second. An important aspect to note in table 2.5.1 is that the other uninformed algorithms, notably IDDFS, BFS, and DFS, are completely unaffected by the directional cost. This is as they do not consider the cost in their pathing and searching method and therefore, present the same values in both directional and normal versions of the same map.


Agent Actions

10 : Up
9 : Right
8 : Right
7 : Up
6 : Right
5 : Right
4 : Right
3 : Right
2 : Down
1 : Right

Uniform Cost Agent, Artificial Delay: 0, Directional Cost: False, Nodes: 45

Agent Actions

12 : Down
11 : Right
10 : Right
9 : Right
8 : Right
7 : Right
6 : Right
5 : Right
4 : Right
3 : Right
2 : Right
1 : Down

Uniform Cost Agent, Artificial Delay: 0, Directional Cost: True, Nodes: 39

Figure 4.2.1. – UCS on Non-directional cost vs Directional Cost.

Otherwise, directional cost as a somewhat negligent impact on informed searches as they ultimately provide the same path to get to the goal, despite the increased costs. An interesting avenue of study would be to allow the informed agents to seek our both goals instead of just one, allowing for a more precise look over the impact of directional cost on any one agent, as at current only one goal is selected as the main goal for the informed agent.

V. Conclusion

Final Thoughts

Given the nature of the robot navigation assignment, there is a lot that can be said about the scratch-built navigation program. Throughout its development, one thing has been rather clear: there is a distinct difference between *searching* and *moving*. There were significant hardships in transposing a few algorithms to both the navigation program itself in a meaningful manner that would be relatively functional and future-proof. Furthermore, the GUI requirement changed the scope of the program significantly due to the increased number of access points required within the agent data itself.

This led to a somewhat large and convoluted program; however, it has achieved its goal of being able to describe various agent search algorithms in an intuitive and educational manner. Notably, the implementation of two uninformed search algorithms as they were both highly unique and were of significant value to for comparison to the informed counterparts. The program was successful in its attempt to compare the values and peculiarities of each algorithm by providing a visual aid in the searching process, notably, the depiction of the complete node tree allowed the dynamic visualisation of "stepping up the tree" in DFS searches, or the breadthwise traverse of BFS.

For the Robot Navigation problem, all algorithms presented pose a space and time complexity issue when faced with significantly larger maps. The most promising algorithm shown was A-Star, providing the most optimal pathway in the least amount of memory and time. Further optimisations to the problem itself would allow for increased pathing speed in very large environments, such as breaking off the maps into "chunks" and applying the A-Star algorithm on the macro level, and afterward, the micro level. This would circumvent the space and time complexity issues provided by extremely large maps within A-Star. Ultimately, the best algorithm is one that is well-designed for the task at hand, be it speed with UCS, memory with IDDFS, or the BFS/DFS in a blind search – the solution is chosen based on the problem, and not the other way around.

VI. References

6.1. Bibliography

URL	Assistance
https://techdifferences.com/difference-between-tree-and-graph.html	A concise description of Tree vs. Graph
https://en.wikipedia.org/wiki/Big_O_notation	A highly detailed webpage outlining the various uses of big-o notation. A good read, not a good reference.
https://www.youtube.com/watch?v=ySN5Wnu88nE A* (A Star) Search Algorithm - Computerphile	A perfect description of A-Star and its uses
https://www.youtube.com/watch?v=5LMXQ1NGHwU Lecture 9 | Search 6: Iterative Deepening (IDS) and IDA*	A lecture on iterative deepening algorithms, including one on IDA*.

6.2. References

Norvig, P., Russell, S. J., 1994, *Artificial Intelligence: A Modern Approach,* Prentice Hall.