Sheffield Hallam University

Faculty of Science, Technology and Arts

Department of Computing

Project (Technical Computing)

[55-604708]

2019/20

Author:	**Benjamin Neil Moore**
Student ID:	**25024749**
Year Submitted:	**2020**
Supervisor:	**Jonathan Saunders**
Second Marker:	**Sergio Davies**
Degree Course:	**Computer Science for Games**
Title of Project:	**Game-based AI pathfinding implementation and optimization**

Confidentiality Required?

NO ☒

YES ☐

Table of Contents

1 - Introduction

As artificial intelligence has developed, a subsequent trickledown effect has seeped into consumer markets. A common application, video games, has incorporated artificial intelligence with continuous annual evolvement. This can be attributed to two main aspects. The first, development of performance increasing hardware, decreasing processing and speeds of computation time. The second, algorithms being researched & developed with techniques being used to better handle artificial intelligence and memory management.

In the Gaming industry, artificial intelligence pathfinding is one of the core fundamentals of a game. Its purpose is to get the shortest route between two points. This can be seen in many genres of games such as first-person shooters, racing simulators, real-time strategy, and puzzle games.

The focus of this project came about when trying to understand what technique should be applied to a real-time strategy game and provide a better grasp of the fundamentals. There are numerous techniques that have been developed that could be used. This project investigates various pathfinding algorithms, drawing comparisons in terms of efficiency and performance within a gaming environment. Another part of the project investigates performance improvements by applying optimization techniques across such algorithms

1.1 - Aims & Objectives

The following aims and objectives have been identified to successfully complete this project.

1.1.1 - Pathfinding Algorithms in Game Engines

Various algorithms are researched and implemented in a game engine to determine the most efficient in terms of speed, optimality, and behaviour in a gaming environment. Tests are conducted in different map scenarios; extracting the data into tables to be analysed and determine the most efficient algorithm. A multi agent pathfinding algorithm is applied to show its efficiency in handling large number of units compared to single agent pathfinding. This is important in strategy games, since the algorithm chosen can affect the number of units that can be instantiated.

1.1.2 - Optimization

Multiple optimization techniques improving the performance of the algorithms are researched. This project applies two of these optimization techniques, one to improve the calculation times and the other to improve path optimality. Tests are conducted, applying the optimization techniques to the algorithms and the data extracted into tables. The data is compared with the same algorithm without the technique to reveal if any significant improvements to performance has occurred.

1.1.3 - Dynamic Map & Pathfinding

The pathfinding and maps are designed to be dynamic allowing the user to apply changes to the map and the pathfinding adapting to these changes, altering its path if required. These changes can be seen with a visual representation that allows the user to see the path behaviour and any modifications made to the map. This feature is implemented to demonstrate the functionality of a real game, allow the user to see the path behaviour calculated by the algorithms and prove it's possible to have all the algorithms be dynamic.

1.1.4 - User Interface & Unit Instantiation

The user interface allows the user to have a better experience when using the application. Giving them the option of navigating around the various algorithms and optimization techniques that have been applied. The unit uses a pathfinding algorithm to generate a path for it to traverse. This is another feature that is used to simulate a real game demonstrating that the algorithms in the project can be used for artificial intelligence game development and provides a visual demonstration of the difference between multi-agent pathfinding and single-agent pathfinding.

1.2 - Project Outline

To develop the application for this project and achieve the desired results there are five key project outline phases:

1. The research phase is to gather material for the project and any prior research that has been conducted in other research papers. Focusing research on the game engines, pathfinding algorithms, optimization techniques and any related information.
2. The design phase is selecting the appropriate methods and techniques that was gathered from the research, explaining why they were chosen and how they are going to be designed and applied.
3. The development phase is producing the application in our selected game engine, discussing how the main parts of the project were implemented and any issues that came along with it.
4. The testing phase conducts tests for each algorithm, the results are then analysed, discussed about and the author providing his thoughts on the outcome.
5. The critical reflection & evaluation phase discusses if the project was a success, critically analysing the whole process of the project from beginning to end with the authors opinions on the success of the project, what skills were learnt and future aspirations.

2 - Research

The following section will investigate various aspects related to the overall objectives. Ultimately contributing to the proposed implementation of the project.

2.1 - Game Engine Analysis

The following section investigates the various game engines that are currently on the market and discuss the qualities they have. The project will implement the solution into a game engine rather than a standalone C++ application to demonstrate it being used in a gaming environment.

Table [1]: Game Engine table comparison.

Descriptor	Unity	Unreal Engine 4	CryEngine	Amazon Lumberyard
Programming language	C# & JavaScript	C++	C++, LUA & C#	C++
Resources, material & documentation	High	Average	Low	Low
Development Community	Large	Large	Small	Small
Access to Source Code	Only reference	Reference & edit	Reference & Edit	Reference & Edit
Graphics	Average	High	High	High

Note: Information gathered from sources (Unreal Engine, 2020) (Technologies. U, 2020) (CRYENGINE | Features, 2020) (Amazon Lumberyard: Features, 2020).

Each engine either uses C# or C++. General implementation of C++ requires a greater know how and advanced coding knowledge compared to the other languages. However, positives include greater freedom to apply advanced techniques to the code. It requires the developer to handle the memory management, which if done incorrectly, can be detrimental. Otherwise, successful implementation provides an optimized solution. C# on the other hand, handles the memory management and provides functionality that restricts what can be done compared to C++ in terms of optimization. Despite the optimization restrictions, it provides greater efficiency in terms of productivity and is a friendlier language to work with.

Unity is the only Game Engine that limits live source code editing. All three engines that allow editing, use C++. This gives the developer more control over their codebase allowing easy alterations and optimization of the engine code to suit their purpose

Both Unity and Unreal have a very large development community compared to CryEngine and Amazon lumberyard. This reflects on the amount of resources, material & documentation that can be found for each engine. It is important to note, that although unreal has a large amount of documentation, the primary base of Unreal's literature catalogue revolves around their visual scripting systems "blueprints" rather than C++ code implementation. These resources are considered less helpful and therefore less valuable in most development cases.

2.2 - Graph Theory

A graph has a set of nodes (vertices/points) and these are all interconnected by edges (lines). Pathfinding algorithms are applied to generate a path between two graph points, a path consists of a sequence of nodes and edges between them (B. Sobota, 2008). A graph uses edges to connect the nodes, an unweighted edge has no value associated with it or all the edges cost the same value making no difference. A weighted edge has a value associated with it.

The undirected pathfinding follows the pattern that the algorithm was designed for and blindly searches for a path. This results in exploring parts of the graph that are not necessary. Directed pathfinding doesn't blindly rush into searching for a path it will have a method to assess the surrounding nodes and will choose the node with the lowest cost. This cost is measured usually by the distance between the nodes and the weighted edge (Graham. Ross, 2003).

"In a weighted directed graph, we might mark a paved road as weight 1 and a twisty forest path as weight 4 to make the pathfinder favour the road. In a weighted undirected graph, we might mark downhill edge B→C with weight 2 and mark uphill edge C→B with weight 5 to make it easier to walk downhill." - (Grids and Graphs, 2020).

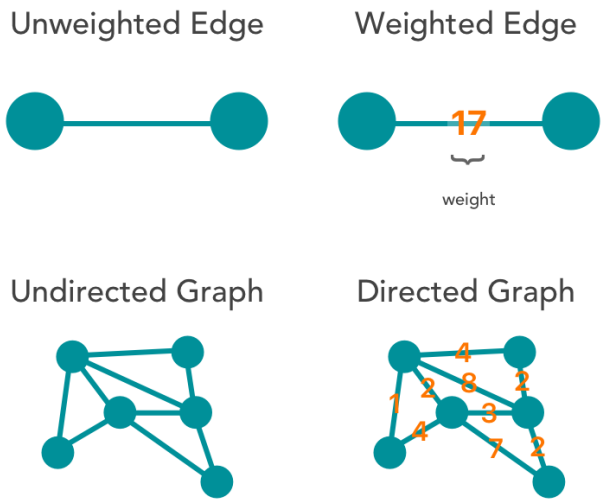


Figure [1]: Directed and undirected graph example (Graph Theory, 2019).

2.3 - Search Space

The search space represents the graph in the game environment, the artificial intelligence algorithms use the search space to calculate a path between two points. When creating a game, deciding on the search space is an important factor that can't be overlooked. Choosing the incorrect one can cause game environment implications such as performance issues, unrealistic movement, and non-optimized paths. The various ways to represent a search space is by using a rectangular grid, quadtree, convex polygons and waypoints (Cui. X & Shi. H, 2011).The larger the search space the more memory it consumes, due to the increased number of nodes and edges in the search space (N. Sturtevant, 2013). On the other hand, a much simpler search space requires far less memory space but runs the risk of inaccurately misinterpreting the search space in the world causing unrealistic movement and non-optimized path (Amit Patel. Map representations, 2020).

2.3.1 - Grid Graph

The grid graph is composed of nodes that are connected by edges displaying a grid pattern. (Abd Algfoor, 2015). Out of the three graphs talked about in this research section the grid search space is the one that has the most nodes. Due to the number of nodes it has, the calculations required for an algorithm will take longer affecting the performance and memory if not handled correctly.

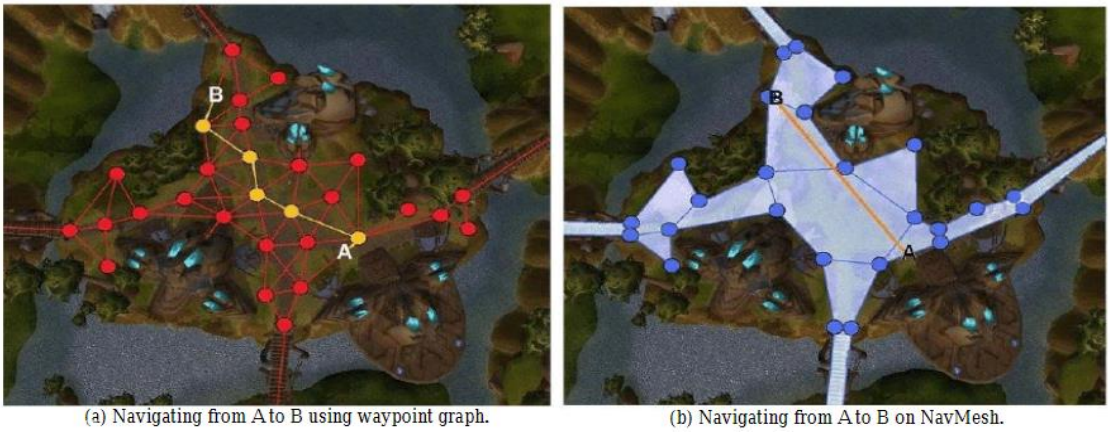
2.3.2 - Nav Mesh Graph

The Nav Mesh graph use meshes that can be represented in different ways, usually as triangles, polygons, hexagons, or points (Kim. H, 2011). When comparing this to the grid graph, it will outperform it due to having significantly less nodes. This reduces the computation time necessary when it is searching for the shortest path. Unity incorporates this approach and has a built in nav mesh that can be used for pathfinding.

2.3.3 - Waypoint Graph

The waypoint graph is the simplest out of the three graphs mentioned. Paths are created by connecting the nodes acting as waypoints (A* Pathfinding Project: Graph Types ,2020). Waypoints can be placed across the game environment giving you the freedom of choosing where to place them so long as there are no obstructions.

Cui. X & Shi. (2011) mention a comparison between a Nav Mesh and waypoint implementation in the game World of Warcraft. Demonstrating the difference in the two approaches, in (a) it uses 28 waypoints to represent possible destinations in the map. While (b) only uses fourteen convex polygons. The movement is more realistic and human, compared to (a).



(a) Navigating from A to B using waypoint graph. (b) Navigating from A to B on NavMesh.

Figure [2]: Navmesh & waypoint graph comparison (Cui. X & Shi. H, 2011).

2.4 - Heuristics

In pathfinding the heuristic is an important part in a weighted search algorithm, using the correct heuristic method can improve a algorithms path optimality. Each heuristic method is designed to estimate a distance between two points. A heuristic method wants to be admissible meaning it will never overestimate the cost of reaching the goal and will always lead to the most optimal path.

2.4.1 - Manhattan Distance

The Manhattan distance comes under numerous names, rectilinear distance, Minkowski distance, L1 distance, taxicab metric or city block distance (Pandit. S, 2011). It calculates the distance between two nodes that are measured along the axes, at right angles (Nagelkerke. T, 2016). The cardinal distance cost is one unit of movement, the diagonal distance cost is two units of movement.

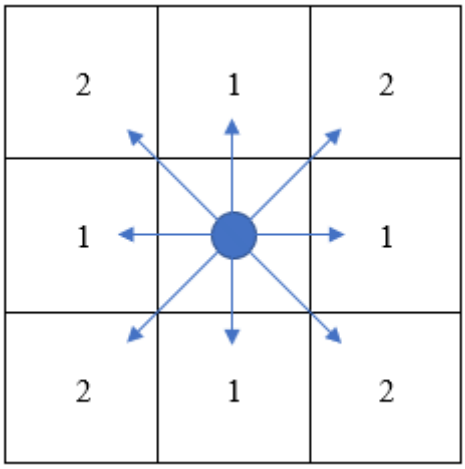


Figure [3]: Manhattan distance.

2.4.2 - Euclidean Distance

This is the ordinary distance between two nodes that one could measure with a ruler (Pandit. S, 2011). It is the straight-line distance between two nodes (Satre Meloy. A, 2019). This method is used when wanting to move in any direction.

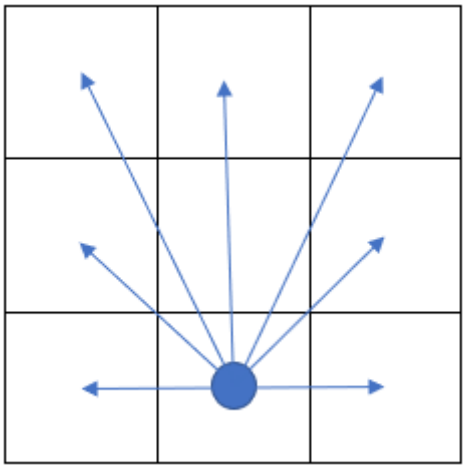


Figure [4]: Euclidean Distance.

2.4.3 - Diagonal Distance - Chebyshev & Octile

The diagonal distance is a broader term name for estimating a distance between two nodes, that allow for both cardinal and diagonal directions. There are different variations of the formula, such as Chebyshev distance and Octile distance. Chebyshev makes all adjacent cells unit of movement equal to one (Euclidean vs Chebyshev, 2012). Octile distance makes cardinal directions cost one unit of movement and diagonal directions cost one point four unit of movement (Zhang. A, 2016). One point four is the square route of two rounded up.

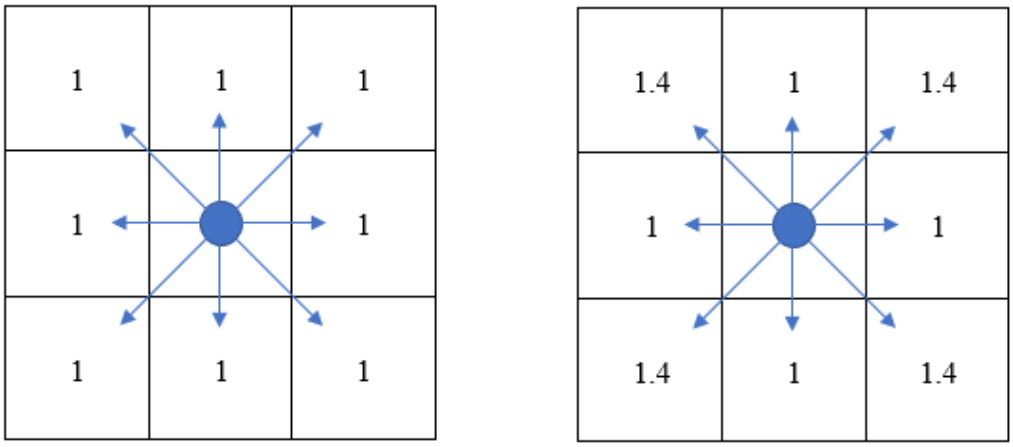


Figure [5]: Left diagram Chebyshev Distance right diagram Octile distance.

2.5 - Pathfinding Algorithms

This will investigate the various algorithms that are currently being used in the game industry and investigate what defines them and the difference between the other algorithms. For each algorithms path behaviour, please refer to **Appendix A**.

2.5.1 - Breadth-First-Search

The breadth-first-search is one of the most basic of the search algorithms. Its main purpose is to attempt to find most optimal path on an unweighted graph. The algorithm in most cases uses a queue as it's data structure, using the principle first in, first out (FIFO) (Anderson. R, (2020). This means the first object placed into the Queue is the first object that will leave the Queue.

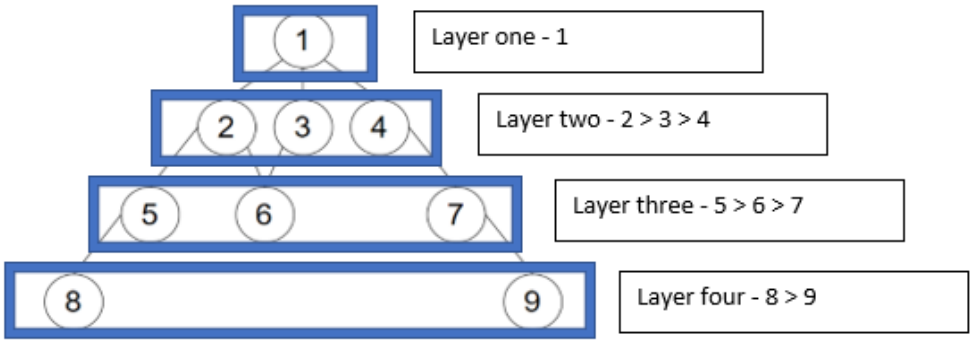


Figure [6]: Tree diagram of exploration traversal for breadth-first-search.

A node is chosen to be the start node, it will then add all the nodes adjacent neighbours to the queue. Once the node has been explored the next node from the queue will be investigated placed in the queue, this cycle will keep happening until there are no more nodes left to search or the path has been found. As the diagram shows once a layer has been completed it will move onto the next layer. (Breadth First Search - HackerEarth, 2020)

2.5.2 - Depth-First-Search

The depth first search is usually compared against the breadth first search, it is another fundamental search algorithm that is used on an unweighted graph. Unlike breadth-first search that uses the queue data structure and follows the principle FIFO. Depth-first-search in most cases uses a stack as its data structure, using the principle of last in, first out (LIFO) (Anderson. R, 2020). This means the last object placed into the stack is the first object that will leave the stack.

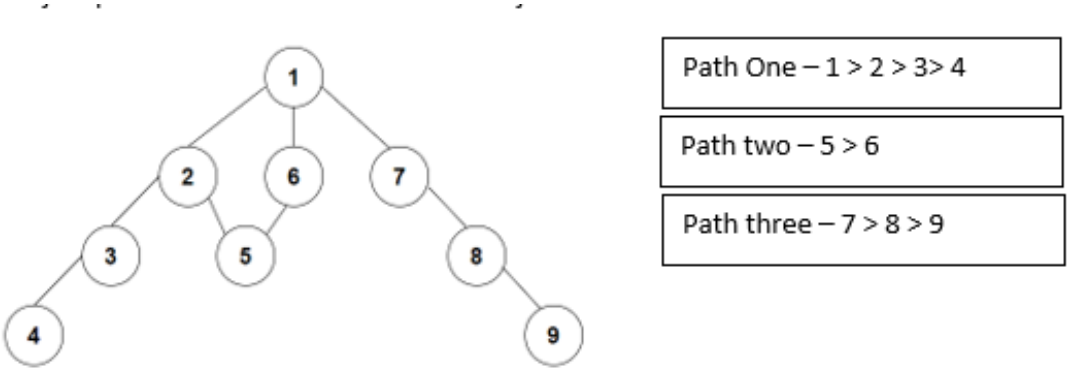


Figure [7]: Tree diagram of exploration traversal for depth-first-search.

A node is chosen to be the first node to be investigated, the nodes adjacent neighbours will then be added into the stack. Once the node has been investigated the next node to be looked at will be popped from the back of the stack. This cycle will keep happening till there are no more nodes left to explore or the path has been found. In the diagram the exploration traversal can be interpreted as once a path is fully explored, it will then go onto the next closest path. (Depth First Search - HackerEarth, 2020)

2.5.3 - Dijkstra

This algorithm is one of the most well-known search algorithms, created by Edsger Dijkstra and published in 1959 (A.M. Turing Award Laureate, 2020). Many algorithms have extended upon from Dijkstra such as A* or uniform search cost. The data structure that commonly used in the Dijkstra algorithm is a priority queue.

The implementation of Dijkstra sets all the nodes in the graph to infinity, it then stores all the nodes from the graph into the priority queue. The chosen start node is given a cost of zero, this will be the first node to be expanded upon. Each adjacent neighbour linked with the source node, will have their cost calculated by adding the start nodes value with the distance calculated between the start node and the neighbour.

$$f_{(n)} = g_{(n)}$$

Figure [8]: g(n) is the path cost of the start node to node n (Norvig. P R, 2002).

If a neighbour already has a cost, then it will check to see if the new cost is lower than the current neighbour cost if it is then it will be overwritten. The next node to be expanded upon is the node with the lowest cost in the priority queue. This cycle will continue till there are no more nodes to explore, completely ignoring if there is a goal node. (Mehta. Parth, 2013)

2.5.4 - Uniform Cost Search

Uniform search cost is a variant of the Dijkstra algorithm that was designed to improve the efficiency of Dijkstra. The two differ as Dijkstra stores all the nodes from the graph into the priority queue and calculates the cost of all the nodes. Whereas, Uniform cost search only stores the starting node into the priority queue and will expand the priority queue from that start node. The same process as Dijkstra will occur but stops the search when the goal node has been found or when there are no more nodes left to explore.

Felner, Ariel. (2011) conducted a test to see the difference between Dijkstra algorithm and uniform cost search, the results were the UCS being much faster.

Table[2]: Result table of DA vs UCS

	Max-Q	Swaps	Time
Single-source all shortest paths			
DA	900,435	15,579,369	13.778
UCS	1,802	8,040,484	8.444
Source-target shortest path			
DA	900,435	7,910,200	7.473
UCS	1,469	3,900,530	4.234

Note: Table from Felner, Ariel. (2011).

2.5.5 - Greedy Best-First-Search

Greedy best-first search algorithm has been adapted from the uniform cost search; it has the same data structure using a priority queue. The difference between them is how the cost is calculated for a node. Greedy Best-first search node cost is determined solely on the estimated distance between the node and the goal node calculated using a heuristic method.

"A function that calculates such cost estimates is called a heuristic function and is usually denoted by the letter h" - (Norvig. P R, 2002)

$$f_{(n)} = h_{(n)}$$

Figure [9]: f(n) is the estimated cost of cheapest path from n to goal node(Norvig. P R, 2002).

In short it will only expand the node closest to the goal and this cycle continues till there are no more nodes to explore or the goal node has been reached. The benefits that greedy best-first provides is the speed it takes to reach its destination, the downside it loses out on path optimality (Felner, Ariel. 2011).

2.5.6 - A* Pathfinding

A* is a best-first search algorithm that is a hybrid of uniformed cost search and greedy best-first search. A* calculates the value of its nodes by using the implementation used in Dijkstra, adding the node cost being looked at with the distance calculated between the start node and the neighbour Then using the implementation used in greedy-best-first search by estimating the distance between the neighbour and the target node (Kuffner, J. J. 2004).

$$f_{(n)} = g_{(n)} + h_{(n)}$$

Figure [10]: f(n) is the estimated cost of the cheapest solution through n (Norvig. P R, 2002).

These two costs are added together, becoming the node cost. It will choose the node with the lowest cost in the priority queue to be expanded upon next. This cycle continues till there are no more nodes to explore or the goal node has been reached. This provides the speed up benefits of greedy best-first search and the path optimality of uniform cost search providing us the flexibility of both algorithms (Cui. X & Shi. H, 2011) (Norvig. P R, 2002).

2.5.7 - Flowfield

Flow field is a multi-agent pathfinding algorithm and steering technique that was designed for efficiently handling large amounts of units (Emerson. E, 2013). The previous algorithms discussed are all single-agent pathfinding algorithms this means each unit needs a path calculated for it, this is very taxing on resources the more units that are in the grid. Flow field helps curb that issue, instead a unit looks at the node it's currently on and that node will influence in which direction it will move to next. This keeps happening till the unit reaches the goal. Flow field can use Dijkstra to calculate all the value of the nodes in the grid and then the integration field will be called to set each node in the grid to point to their neighbour with the lowest cost value. (Flow Field Pathfinding, 2020) (Durant. S, 2013)

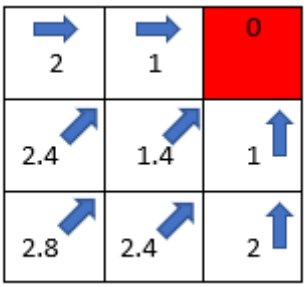


Figure [11]: Flowfield diagram

2.6 - Optimization Techniques

The following section will explore certain optimization techniques that can be applied to enhance efficiency and overall performance.

2.6.1 - Heap/Priority Queue

The binary heap or priority queue is a data structure that is meant to order the nodes into a specific order, bringing the nodes with the highest priority to the front of the data structure. The priority can be determined by the way it is designed; a maximum binary heap will prioritise the highest value, a minimum binary heap will prioritise the lowest value.

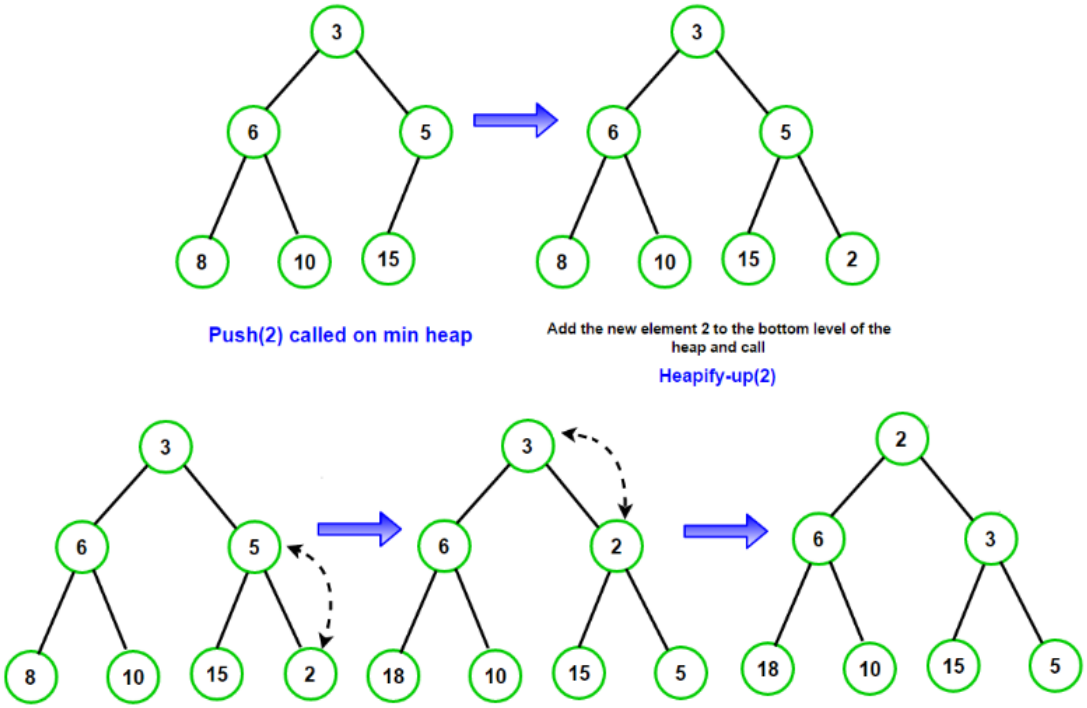


Figure [12]: Min heap process diagram (Techie Delight, 2016).

This works by adding a node to the minimum binary heap, it will have its cost compared with the node above it. If the value of the node is lower than the node it is being compared to, they will swap places. This keeps occurring until it meets a node with the same value, or the node has a lower value than it.

2.6.2 - Pre-Calculate Every Single Path Floyd-Warshall Algorithm

This optimization is done by pre-calculating every single path that can be achieved in a search space and storing the values in a lookup table. In English speaking circles this is called Floyd-Warshall algorithm, while in Europe it is better known as Roy-Floyd (Rabin. S, 2013).

All that is required to find the path needed is by looking it up in the lookup table. This method has astronomical consequences on the memory but there are techniques that can be applied to limit the repercussion. The improved performance will far outweigh the memory expenditure, becoming a very viable optimization technique.

2.6.3 - Heuristics

For A* to have optimal behaviour it needs to have an admissible heuristic. This means the estimation of the heuristic cost, from the node to the goal node must never overestimate the true cost to get the most optimal path (Rabin. S, 2013). It is not all doom and gloom by making it non-admissible although losing out on a non-optimal path only slightly, it will expand fewer nodes leading to an increase in computation time. A small amount of overestimating can lead to huge performance gain with very little noticeable non-optimality.

Kumar, P (2004) conducted a test that used a A* algorithm that used the Manhattan distance heuristic and another that used the Manhattan distance heuristic along with an overestimated heuristic. What occurred was the latter speed had improved greatly and the path although not optimal still looked like a viable path.

This may seem obvious but choosing the correct heuristic can also lead to performance gain. The Manhattan distance is a poor heuristic to use if your search space allows both diagonal movement and forward movement since it will overestimate the heuristic when going diagonal. In this case you would want to use the octile since it is an admissible heuristic that takes diagonal distance into consideration.

2.6.4 - Multithreading/ Parallelizing

Multithreading is the ability to run multiple threads synchronously. The hardware this occurs in is the central processing unit (CPU), the more cores you have the more beneficial multithreading will be. This allows you to run code or tasks at the same time, this can be very beneficial in pathfinding. Take A* and Dijkstra algorithms for example, the issue they have is that the execution times take much longer the bigger the map becomes especially when it has to examine each neighbour node, beginning from the start node all the way to the goal node.

Mariam Arshad (2017) overcame this issue by applying parallelization. They did this by splitting the pathfinding problem into sub parts where search space is divided among multiple threads. Allowing multiple cores to work synchronously to find the best path. This brought the computational time down. Depending on the number of cores you have, the more cores you have the bigger the speed up will be.

Table[3]: Parallelization results

Size of grid	Sequential execution time	Parallel execution time	Speedup
2000*2000	72.895	16.752	4.351
3000*3000	173.669	52.725	3.294
4000*4000	396.468	123.189	3.218

Note: Table from (Mariam. Arshad, 2017).

2.6.5 - Hierarchical Pathfinding

One of the biggest drawbacks when implementing A* or Dijkstra in a grid is the number of nodes it must go through when searching, especially when the size of the search space is very large. Hierarchical pathfinding helps lower the number of nodes, that need to be searched. This works by splitting the grid into sections. These sections represent a certain number of nodes in the search space (Duc, L, 2008). Once the start and end node have been placed in the grid, the higher-level aka the sections will have a path calculated. Once a path has been calculated on the higher level, it will move onto using those sections that are in the path, searching another path between the lower levels aka the node in the sections.

This is method that can be a double-edged sword, it provides a much-increased speed in searching the nodes but loses the optimality in its path. Sacrificing precision for a much-increased performance boost. However, the loss in path quality is negligible and still achieves close to the most optimal. Hierarchical pathfinding would be beneficial on a dynamic map. Instead of having to update the whole grid, only a section of the grid would need to be updated. This will mean only applying changes to the nodes in that section.

Botea. Adi & Müller (2004) applied hierarchical pathfinding to A* called HPA*. They conducted a test comparing a highly-optimized A* against the HPA*, the results showed a great reduction of the search effort needed compared to the highly-optimized A*. The HPA* was shown to be up to 10 times faster, while finding paths that was within 1% of optimal.

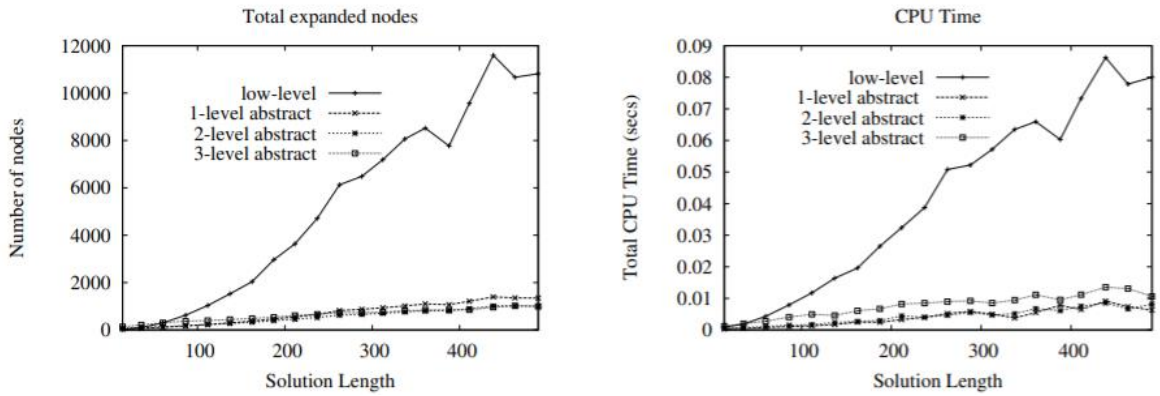


Figure [13]: Low-level A vs hierarchical (Botea. Adi & Müller, 2004).*

3 - Design/Methodology

This section will investigate the techniques and method that have been chosen for this project and explain why the choice was made. Providing relevant information for the design or the approach that will be taken for the design.

3.1 - Game Engine

The deliverable will be developed using the Unity game engine. This comes down to several factors. It uses C# which limits code optimization but does provide a much friendlier language to produce a more finished product. There are plenty of resources and materials that can be found to be used as reference. Unity doesn't allow access to edit the source code but for what the project is trying to achieve it is not required. Visual studio provides a unity extension that allows for debugging of unity scripts within the IDE making It ideal for debugging. Learning how a new game engine works and the functionality compared to other engines takes up a lot of time, the author experienced this when working on the unreal engine. Instead of focusing time on trying to understand how the engine works, the focus is to spend time on understanding AI algorithms, it is the core of the research. Unity provides what is needed to sufficiently achieve all the target aims in this project in the given timeframe.

3.2 - Supporting Software

There will be two supporting software being used to help develop the Unity application. One is visual studios integrated development environment, this provides supports for a whole suite of built in languages such as C, C++, C# and F# to name a few. It supports other languages like JavaScript or Node.js but you would have to install the service language yourself (Overview of Visual Studio. 2019).

Visual Studio was chosen for several reasons, the visual studio development community is enormous, it's provided plenty of documentation and has excellent user support in case of running into any bugs. There are a wide range of extensions created by Microsoft and the community. Two extensions that will be used in the project is the GitHub extension this allows a person to connect their repository located at GitHub and push, pull, or apply any changes through the visual studio IDE. The unity extension allows for a person to debug their scripts that they've created, allowing them to pinpoint any issues going wrong within the code.

The second is the GitHub repository, making sure that there is always a backup of the project. If any changes occur that break the current project build, then it can be reverted backwards to a previous working build.

GitHub was chosen due to its extension for visual studio allowing anyone to push code, pull down code, sync code and create a new repository all in visual studio all that's required is to sync a GitHub account with visual studio. This Makes it a lot easier to access with having to install a third-party programme. GitKraken would have been the second choice but unfortunately doesn't allow people to connect to their private repositories on GitHub without buying premium.

3.3 - Architecture & Class Design

This project will be built off the architecture & class design provided by Wilmer, Lin (2020) in his basic tutorial and has allowed permission to expand upon it for my dissertation, refer to **Appendix I**. This allows the focus of the project to be spent more on the algorithms, dynamic aspects, unit, and user interface. Requiring less effort to be spent on the architecture & class design.

The architecture it uses is the MVC approach meaning model, view and controller. The model are the classes where the data is stored, the view will be the visual representations of our graph in the unity game engine and the controller will be what interacts with the model to manipulate any data or changes to the view. The controller will come and call the relevant functions from the other classes to apply the changes.

3.4 - Min heap and Heuristic

The minimum binary heap will be implemented as one optimization for the algorithms. Many papers and articles mention that the commonly used data structure for a weighted algorithm is to use a priority queue/min heap. The min heap will be built off a list data structure, implementing the necessary functionality to produce a min heap. It will also require an interface IComparable to be used. It is what forces our data type to have the implementation of the CompareTo that would be necessary to compare values.

The second optimization is applying different heuristic methods, demonstrating the difference they can have on each algorithm in terms of path optimality and speed. The heuristics will follow pseudocode and code implementations from Amit Patel Heuristics (2020). Implementing, Manhattan, Euclidean, Chebyshev, Octile and an alternative method for diagonal distance designed by Patrick Lester (2020). All heuristic methods researched will be implemented providing the opportunity to see the difference between the heuristics and have a better understanding of them at the end.

These optimization techniques will only affect the performance of the algorithms of Dijkstra, greedy best-first search and A*.

3.5 - Pathfinding Algorithms

In the research phase various implementations and material was investigated that provided information to create pseudocode for these algorithms. The pseudocode will be followed when implementing the algorithms into the development of this project. The pseudocode has been altered for all algorithms so that they will work with a list that will make implementing the min heap easier since it's going to be built off from a list. All the algorithms researched will be implemented or attempted this will provide a better understanding and show the difference between each algorithm. For pseudocode implementations refer to **Appendix B**.

3.6 - Dynamic Map

The search space chosen for the project will use the grid since it will be able to demonstrate larger improvements in performance when applying any of the optimization techniques.

The first map created will be one that has obstacles in place that the unit will have to traverse around. The second map will be a maze, this is to show the benefits for depth-first-search, the third map created will be a terrain map to show the difference between weighted and non-weighted algorithms.

The user must be able to apply changes to the map in runtime letting them add in their own terrain which will be done in the UI. The pathfinding must be able to detect these changes in case a better path has opened or its path has been blocked.

3.7 - User Interface

The UI must allow the user to have an enjoyable experience when using the unity application. Mouse controls will be applied to make changes on the map, right clicking will change the tile to a goal node, left click will change the terrain based on the terrain dropdown and the middle mouse button will instantiate more units on the map. There will be a drop down to allow the user to select the algorithm they would like to use and an input box for them to manually add a position for the goal node. There will be a console log on the screen to allow the user to see information from the algorithm.

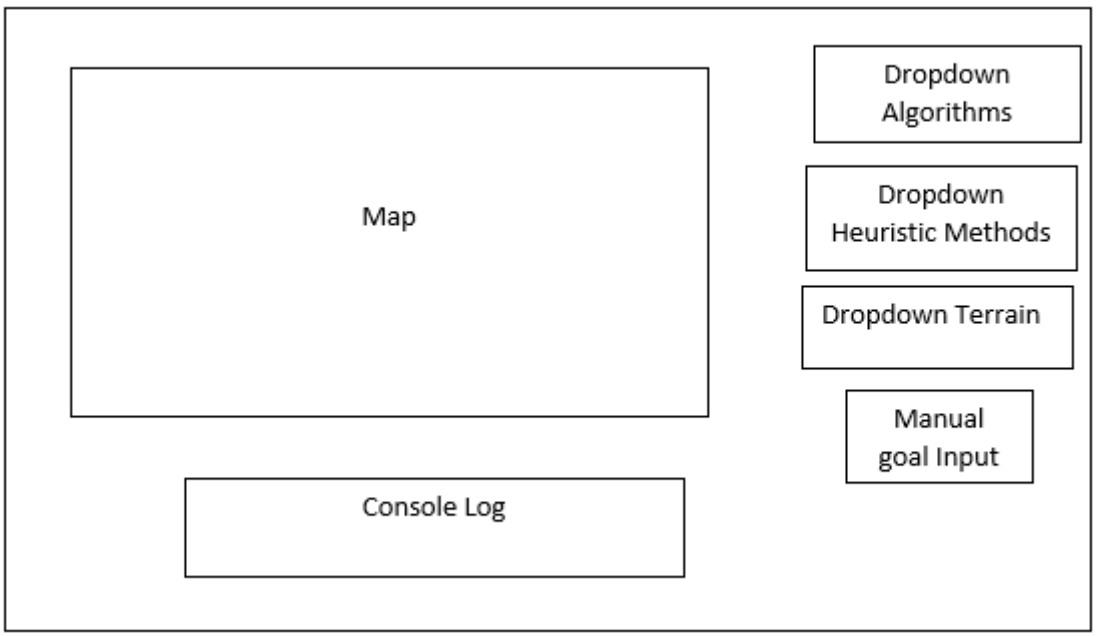


Figure [14]: UI interface design for our scene.

4 - Development

The development section will talk about the key parts in the development phase and what was the outcome and how they were implemented.

4.1 - Map Generation & Map design

In the end four maps were developed the first map is a labyrinth with obstacles around the map, the second map is a maze, the third map is a terrain style map and the last map is a custom map that lets the user create their own map.

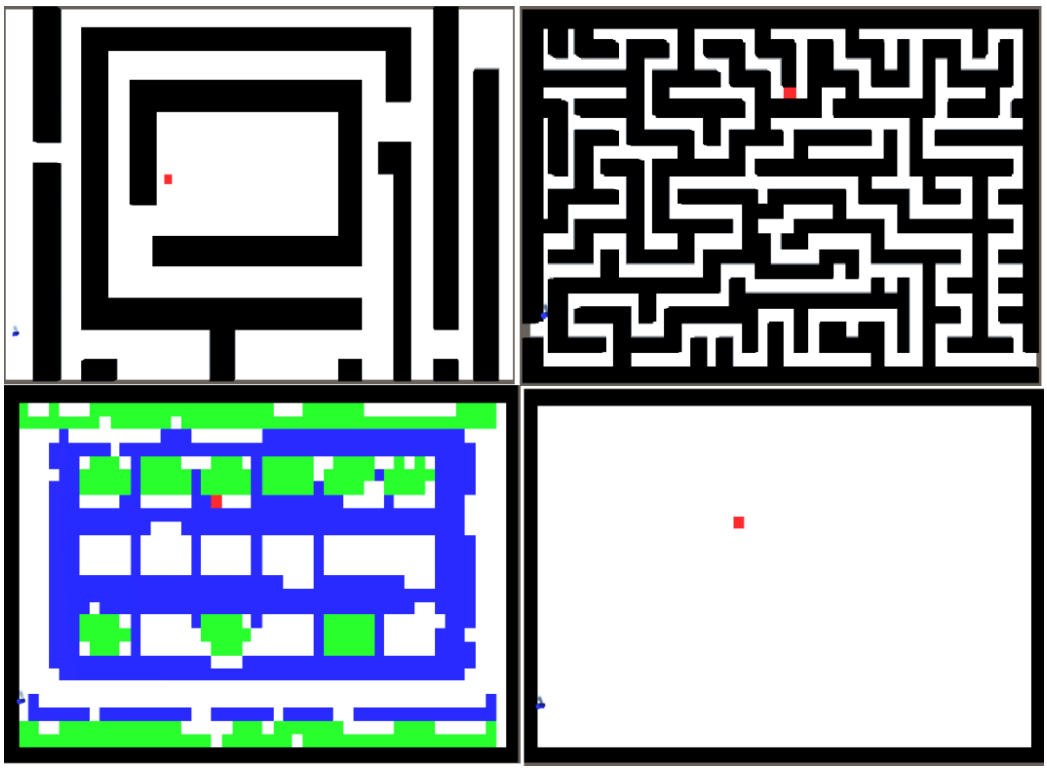


Figure [15]: Top left Labyrinth, top right maze, bottom left terrain and bottom right custom map.

For the map to be generated each controller class has a start function, that will call the relevant functions from the MapData class, GridManager class and GridVisualisation class. MapData has functionality that allows us to generate what our map design will be by reading from a PSD file that is a texture. GridManager will then use the data provided by MapData to instantiate our nodes for our grid. This grid of nodes will then be passed onto Grid Visualisation Class that will instantiate game objects to represent our nodes in the scene. Code implementations for this section, refer to **Appendix C.1.**

4.2 - Visual Aid

The visual aid was developed so that the user can see how the path was generated, the different terrain types and the goal. In flow field it shows the flow of each node using arrows directing you to the goal. The basic pathfinding algorithms show the path of the unit, the nodes in the open list and the closed list each one a different colour.

Figure [16]: Basic pathfinding visuals left and flow field visuals right.

Every game object in our scene is a node visualisation prefab, it is an empty game object that has three game objects attached. This is the tile, the wall and the arrow. The node visualisation script is attached to the empty game object and provides the functionality that allows us to alter the three game objects.

To change the colour of a node, the material of the tile game object is retrieved and using the colour of the material it's altered to the colour specified. For the cube and arrow, these objects are activated or deactivated depending on if they are required. The arrow is only used in the Flowfield map and allows the user to see which direction the flow is going; this means the arrows had to point in the direction of the Nodes parent. This was done by working out the direction the parent was, getting the quaternion and applying it to the arrows game object rotation.

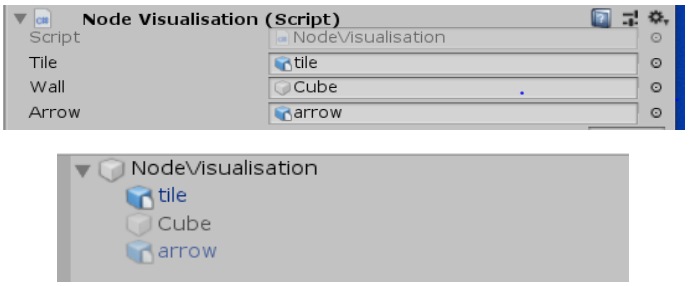


Figure [17]: Node Visualisation prefab.

Our scene controller and pathfinding class call upon functionality in the GridVisualisation class that holds the array of visual nodes. It provides functionality that can allow us to cycle through the array or choose a specific visual node to alter, calling upon the required functions in the node visualisation class. It provides functionality to flush any colour changes from the previous pathfinding search or change in terrain, resetting the colours to the node type. Although this works well it is highly

inefficient since it is resetting the whole map each time, especially when it's called only for a single node being changed. Code implementations for this section, refer to **Appendix C.2.**

4.3 - Dynamic Map/Pathfinding

One of the deliverables for this project was to make the map and the pathfinding dynamic. This was developed successfully; the implementation allows the user to alter the maps terrain in runtime and the unit will be able to recognise the changes and adjust if an object has appeared in its path or a better path has opened.

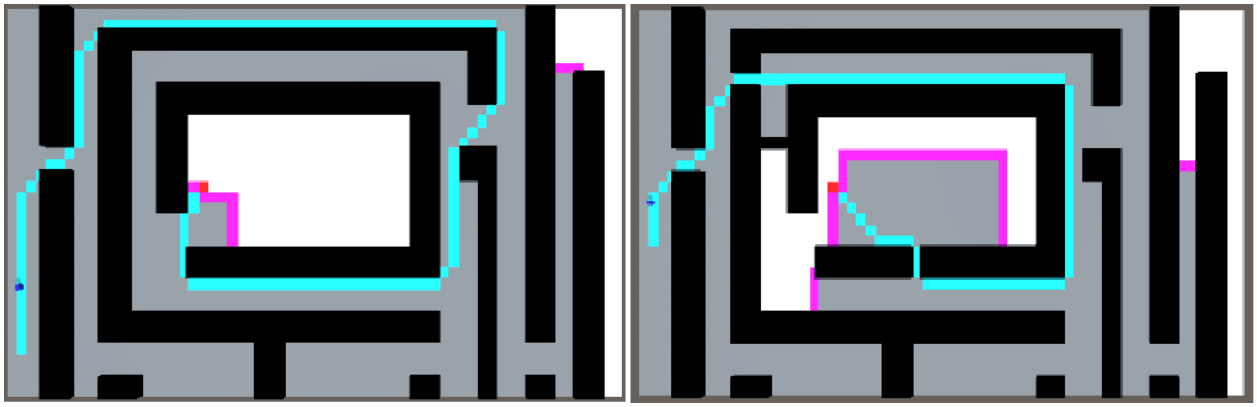


Figure [18]: Dynamic map and pathfinding for basic pathfinding algorithms.

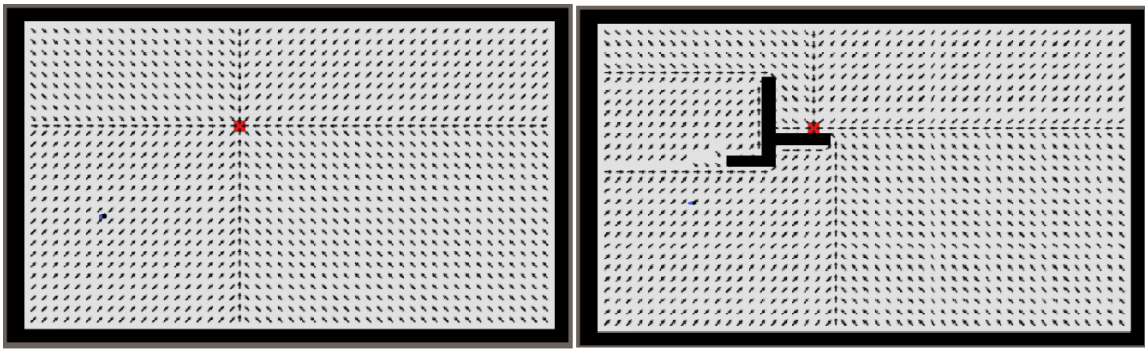


Figure [19]: Dynamic map and pathfinding for Flowfield algorithm

For our map to recognise what visual node in our scene has been clicked a raycast was implemented to hit the game objects in our scene. A change had to be made to the tile game object and a mesh collider added for the raycast to hit the object instead of going through it. This allows us to detect the correct visual node the user has clicked allowing us to call the required functionality.

If a left mouse click has occurred this alters the terrain type of that visual node changing it to the terrain that is selected in the drop-down list. Once the changes have been applied it will call the functionality from the unit class that will recalculate the path of each unit in the scene. If a wall has been placed, it will only recalculate if the wall is in the path of the unit.

For Flowfield I had to implement the recalculate functionality differently, since the units rely on the node and not a list of paths. I had to recalculate the entire grid of nodes so that whatever changes occurred updates on the Flowfield, resetting the values of the unit and the nodes.

If a right mouse click has occurred, the clicked visual node will change to our new goal position and the previous will be set back to a floor. When this occurs Flowfield will update the map after it's changed the tile while pathfinding will recalculate the path of the units after it's changed the tile.

If a middle mouse click has occurred, the clicked visual node will have a new unit instantiated above it and their path calculated. Allowing for multiple units to be in the scene and moving as soon as they spawn. Code implementations for this section, refer to **Appendix C.3.**

4.4 - User Interface

The user interface evolved as the development went along, going past the initial design. This was developed to give the user a better experience. There is a main menu letting you choose what map you would like, map one being the labyrinth, map two being the maze, map three being the terrain map and a custom map that allows you to make your own map design. The user can select the pathfinding they want to do choosing between Flowfield, minheap implementation or basic algorithms. Clicking the buttons on the menu will load the scene associated with that button.

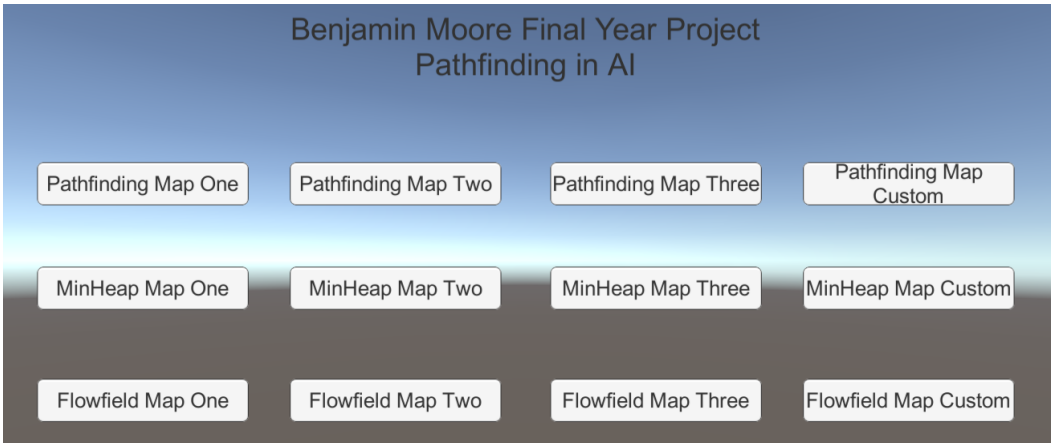


Figure [20]: Main Menu.

The Pathfinding and Minheap scenes have the same user interface that allows for the user to change the algorithm, heuristic method and terrain using a dropdown list. They can input their own co-ordinates for the goal and unit position. Apply their own custom heuristic distance to replicate or demonstrate a heuristic. All changes made by the dropdown or the inputs alters a specific variable in the scene controller class.

The user can toggle the visual aid on or off making sure it happens instantaneously. Every part of the script that changes the visualisation of the map has an if statement that corresponds to the bool Pathfinding Visual Aid. When it's been toggled to false, meaning it's turned off we reset the grid and only have the goal node colour and terrain colours showing.

The console log displays information of the algorithm sent back from each specific unit, adding it to the text box of the console log. It has a scroll wheel that allows the user to go back to see previous pathfinding runs. If there is more than one unit on the map, the console log will be cleared each time

a new path is calculated otherwise it will crash if it passes a certain amount of text. There is also a button that takes you back to the main menu in case the user wants to try the other maps or algorithms.

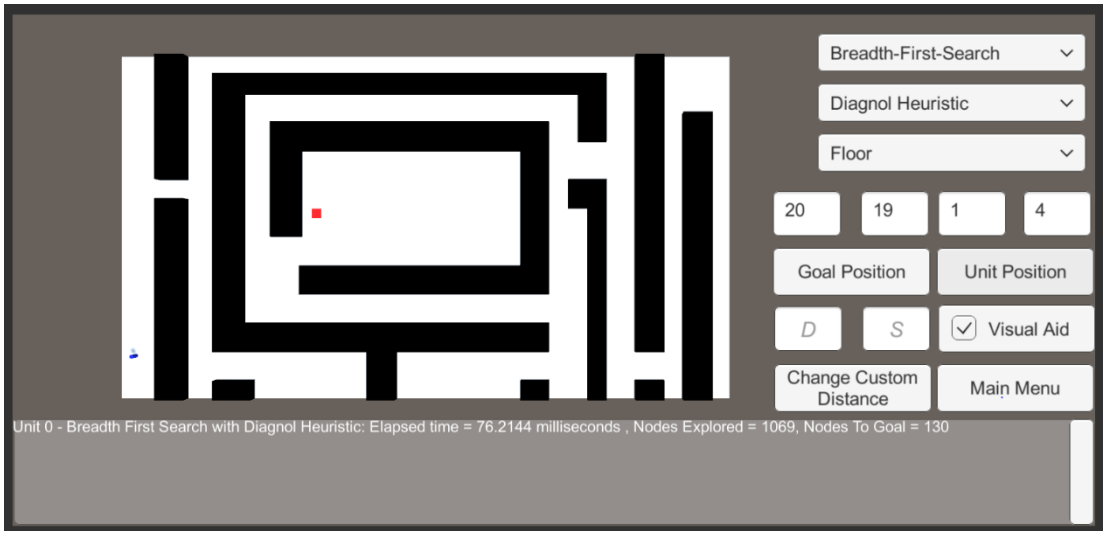


Figure [21]: User interface for pathfinding & minheap scenes.

Flowfield has similar functionality except custom distance and the algorithm dropdown have been removed. Flowfield only relies on one algorithm and all the heuristic methods is not needed, octile distance would have sufficed. I kept the drop down since it may be needed in the future and it does not harm to have it there. Code implementations for this section, refer to **Appendix C.4.**

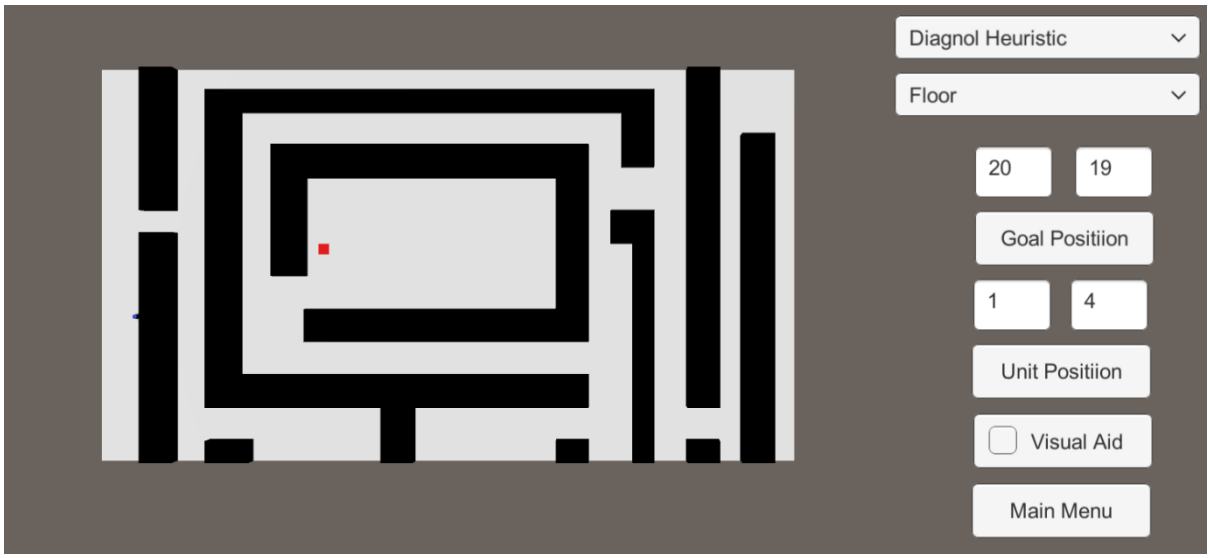


Figure [22]: User interface for Flowfield scene.

4.5 - Unit & Unit Movement

The unit development ensures that an object representing the unit can traverse across the map by following the path that had been calculated for the specific unit. There were three prefab units that got created one for pathfinding, MinHeap and Flowfield. Each prefab has its own unique unit script attached to the cylinder game object. The minheap unit and basic pathfinding unit have the same implementation, both having a child empty game object with a pathfinding script attached. They have separate prefabs due to the pathfinding scripts being different classes. Flowfield doesn't have this game object due to its implementation being different.

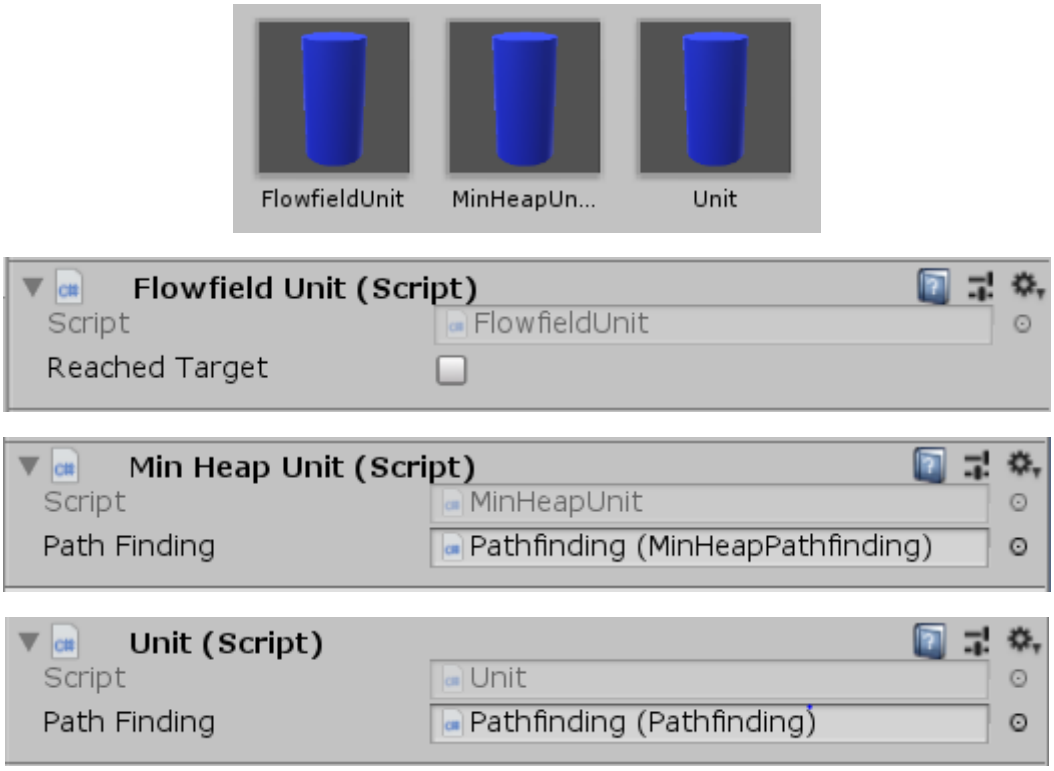


Figure [23]: Unit prefabs and scripts.

When a path for a unit needs to be recalculated or calculated it calls upon function UnitFindPath provided by the unit class. Inside this functionality it calls upon the FindPath function in the pathfinding class to generate a path and message for the unit. When this has been generated it will be passed back to the unit class and placed into a global List and string.

The movement function MoveUnitAcrossPath will then be called in the unit class, this transforms the unit position towards the node that is currently being looked at. When that node has been reached it moves onto the next node in the list till it gets to the goal node. To get it to keep cycling through the functionality till it reaches the goal I use a Coroutine and IEnumerator for each individual unit.

Flowfield does not need its path calculated or messages being sent back. The Flowfield controller calls upon UnitMovementStart passing the node the unit is above through. This is passed onto the movement function MoveUnitToNode in the Flowfield unit class where it will then find the parent of that node and transform the unit position towards the position of the parent node. It will set the parent and current node bool unit above to true to prevent other units heading towards it. Code implementations for this section, refer to **Appendix C.5.**

4.6 - Minimum Binary Heap

The project originally used a list to create the MinHeap data structure but was discovered that using the functionality that came with the List to create the MinHeap behaviour proved to be slower. Moving away from a list, the MinHeap was built from an array instead, combining two different approaches from Grishechko, E. (2019) and Lague. Sebastian, (2019). The array provides better memory management and less functionality compared to a list.

The MinHeap pathfinding class is where the MinHeap data structure gets instantiated and its functionality used. It uses an array that must be provided a data type and array size, the Node class is our data type and the array size is the number of nodes in our grid.

The MinHeap data structure incorporates the same function naming convention as the list. When it came to add the MinHeap to the pathfinding class all that had to be changed was the data structure, changing from a list to a MinHeap.

The MinHeap data structure makes sure that any data type used for the MinHeap provides its own implementation of certain functionality, in our case this is the Node class. The first method allows us to compare the cost of the current node and a node passed through to determine which has a higher, equal, or lower value. The second is a method that allows you to set and get the index position of the node in the MinHeap.

The functionality provided by the MinHeap class allows for us to produce our MinHeap behaviour. When a node is added to the MinHeap, it is put at the back of the MinHeap and is compared with the node above it, using the compare implementation from the node class. If the new node cost is higher or the same as the node it is being compared with it stays where it is but if the cost is lower, it will swap positions with the node.

When a node is removed from the array it takes the node at the front of the MinHeap and places it into a temp variable to be returned through the function. It takes the node at the bottom of the MinHeap and places it at the front. It will then be compared with the node below it and if it's got a higher cost value, they will swap positions, if it equals the same or has a lower value it stays at its position.

There is functionality that allows for a specific node to be checked to see if it's currently being stored in the MinHeap, returning a bool and a function that provides the current size of the MinHeap. Code implementations for this section, refer to **Appendix C.6.**

This has caused a problem when using the Chebyshev heuristic, due to how our neighbour list prioritises nodes in the straight directions first, the nodes in the diagonal directions will be last so when we take from the bottom, we now priorities the diagonal direction instead of the straight. If the values equal the same, then it will stay where it is. Therefore, the path when using MinHeap becomes skewered and appears inaccurate, **Appendix G**.

4.7 - Heuristic

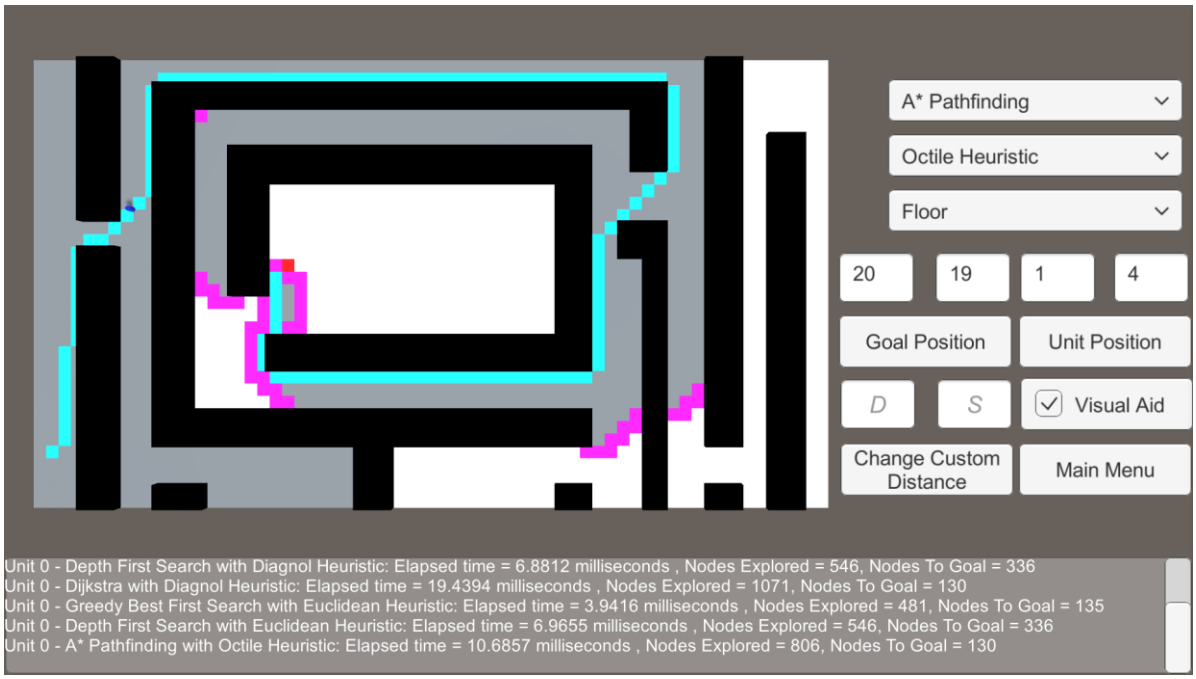
Each heuristic was developed following the implementations mentioned in **section 3.4** Due to having multiple implementations of heuristic methods, a drop-down list was created in the user interface so that the user can select the heuristic method they would like to use. The value chosen from the drop-down list is then passed through the function to the switch statement and selects the method that matches the value.

Each heuristic is designed so that it can return an estimated distance between two nodes that have been passed through. This is calculated by the heuristic method and then is returned through the function.

The custom distance was added last minute, it's the same implementation as Patricks diagonal distance but unlike Patricks where the values are set, it was designed so that the values can be changed in the user interface in the scene. Allowing to better explain the heuristics in the demonstration of the deliverable and allow the user to play around with the values. Code implementations for this section, refer to **Appendix C.7.**

4.8 - Basic Pathfinding Algorithms

All previously mentioned pathfinding algorithms were successfully implemented. The pathfinding class generates a path for our unit and provides the information for the algorithm that is seen in our user interface.



Figure[24] - Path generated and information provided by the pathfinding class.

There are two separate pathfinding classes for the basic algorithms, one applies the MinHeap data structure and doesn't implement the best & depth first search. The other uses a list but implements all algorithms. All the pathfinding algorithms follow the same code implementation.

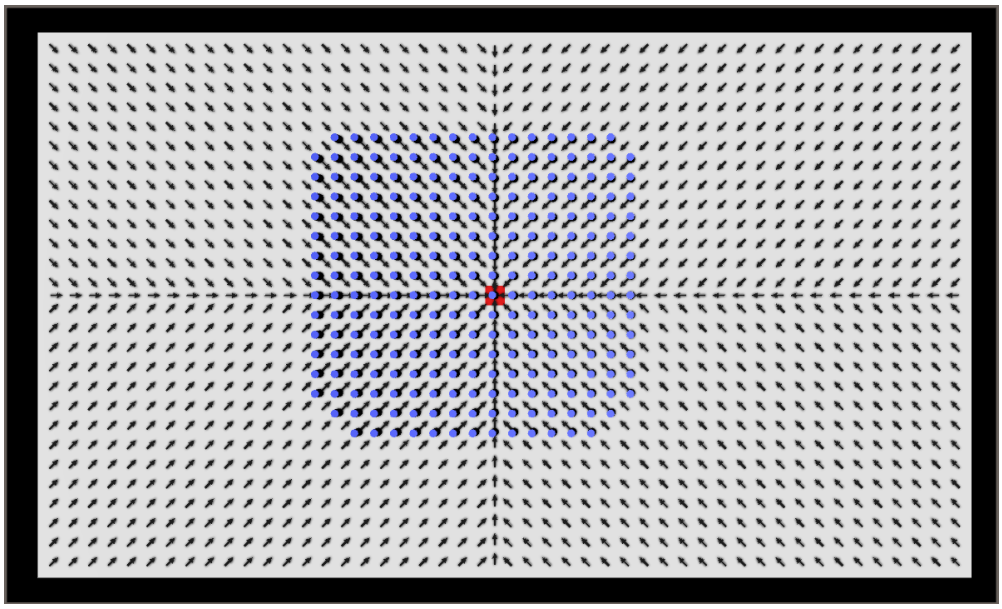
Both these classes have a function called FindPath that is referenced by the unit. The first part that occurs is making sure to reset all nodes parents and cost values back to default, clearing the list/minheap off previous data.

Using the data passed through the function it retrieves the start and goal node, chooses the correct pathfinding algorithm from the switch and the chosen correct heuristic method. It will keep looping through the algorithm till it finds the goal node in the list/MinHeap it will retrace the path, store it in a list and send it back through the function. It will also send a message back through the function providing information of the algorithm if it was successful or a fail message if it was unsuccessful.

Issues arose with the Depth first search in which the path would choose diagonal directions first and not use straight directions. This is due to the ordering of neighbours in the list. Taking neighbours from the end of the list resulted in reprioritising diagonals. Overcoming this, the Neighbours List is reversed so when it takes from the back it chooses a straight direction instead of diagonal direction. Code implementations for this section, refer to **Appendix C.8.**

4.9 - Flowfield Algorithm

The Flowfield algorithm was implemented to allow multiple units to converge to a point on the map efficiently. This was implemented successfully. The Flowfield scene controller calls upon the function FlowfieldPath in the Flowfield pathfinding class, it calculates the cost of each node, determining the parent and the direction the arrow should be facing on each node.



Figure[25]: Flowfield algorithm working on custom map.

Every time this function is called it resets the nodes back to their default values and clears any data left in the list that was left from a previous call. Not doing this will result in the Flowfield not working correctly. It then expands the goal node that was passed through the function and the costfield expands from this node.

The Costfield is the same implementation as the original method of Dijkstra, calculating the cost of each node in the grid and halting when all the nodes have been explored.

The integration field goes through all the nodes in the grid and sets the node parent to the neighbour with the lowest cost value. This determines which direction the unit should be going when heading towards the goal. The arrows on each node are calculated to point towards the nodes parent. The function makes sure to set the Node Parent to itself, this way if all the neighbour nodes are blocked or the other values are too high meaning, we're going backwards then point to yourself.

Another issue was preventing the node from pointing to a node that has a unit currently going towards it or is currently above it. This was to prevent units from colliding into each other and moving onto the same node, the issue can be seen in the basic pathfinding algorithms. If the neighbour node that is being looking at has the bool unit above checked to true, then don't use it since a unit is already heading in that direction or is already above it. It will then use the next lowest value neighbour node, till it can't find a lower value pointing to itself. Code implementations for this section, refer to **Appendix C.9.**

There are two issues with the Flowfield implemented, if the goal is in the middle of a 4 x 4 sea terrain section of the map the units will stop on the outskirts of the sea due to it having a lower value. Potentiality there is a chance of the units getting stuck due to being boxed in.

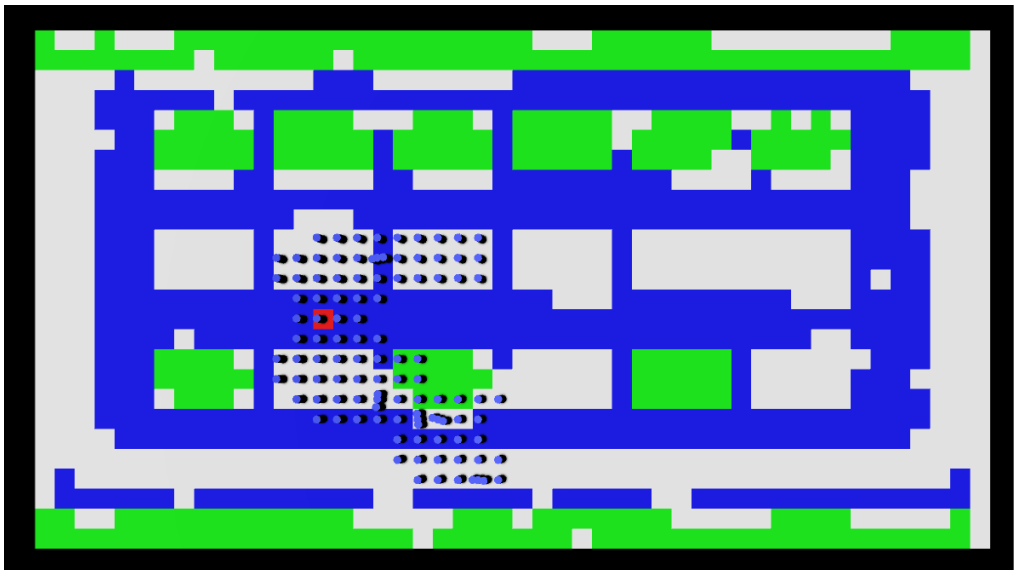


Figure [26]: Issue on terrain type map and units getting stuck.

5 - Testing

The testing section gathers the test data produced by the application and an analysis on each map will be done. Then a conclusion to determine the heuristics or algorithm that should be used in different situations going by the data provided.

5.1 - What is Being Tested?

A part of the deliverable was to determine which pathfinding algorithm is the fastest and most consistent. On top of applying optimizations to increase the efficiency of the algorithms. Implementing different maps allows for different results to appear, due to some algorithms better performing in different situations than others.

5.2 - How Were the Tests Conducted?

Each pathfinding algorithm and Heuristic method was run over 11,000 times the first 1,000 results being discarded. This was done to have more accurate results since it takes time for it to warm up. The three parts of the algorithm that was used was the time it took to complete a path, the number of nodes it took to explore and the number of nodes it took to find a path. The average of the 10,000 runs will be placed into a table.

5.3 - How Will the Test Data Be Displayed?

The parts of the table shown in green are the best results and red would be deemed as the worst results. Orange is classed as a special case and will be talked about. The test data is matched up with the algorithm used and the heuristic used.

5.4 - Map One Labyrinth

5.4.1 - Table [4]: Time it takes to find a path.

Map One - Time it takes to find a path	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	20.84
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	8.08
Dijkstra	21.00	21.14	21.06	20.89	21.02	N/A
Greedy-Best-First-Search	4.76	5.03	4.56	4.57	4.64	N/A
A* Pathfinding	11.98	12.19	12.79	14.91	11.94	N/A
MinHeap - Dijkstra	20.75	21.04	20.95	20.74	21.02	
MinHeap - Greedy-Best-First-Search	4.39	4.72	4.30	4.03	4.36	
MinHeap - A* Pathfinding	11.86	10.48	12.50	13.69	11.79	

5.4.2 - Table [5]: Number of Nodes explored.

Map One - Nodes Explored	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	1069
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	546
Dijkstra	1071	1079	1072	1069	1071	N/A
Greedy-Best-First-Search	490	506	481	477	490	N/A
A* Pathfinding	806	808	830	892	806	N/A
MinHeap - Dijkstra	1071	1082	1072	1067	1071	N/A
MinHeap - Greedy-Best-First-Search	489	504	481	477	489	N/A
MinHeap - A* Pathfinding	803	756	827	868	803	N/A

5.4.3 - Table [6]: Number of nodes in path to goal.

Map One - Nodes to Goal/Path	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	130
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	336
Dijkstra	130	132	130	130	130	N/A
Greedy-Best-First-Search	135	135	135	137	135	N/A
A* Pathfinding	130	132	130	130	130	N/A
MinHeap - Dijkstra	130	132	130	130	130	N/A
MinHeap - Greedy-Best-First-Search	135	135	135	137	135	N/A
MinHeap - A* Pathfinding	130	135	130	131	130	N/A

5.4.4 - Analysis of Result.

Overall, the slowest time is Dijkstra using the Manhattan heuristic taking the average time of 21.14 milliseconds each cycle. The fastest time is the greedy best first search using the Chebyshev heuristic taking 4.03 milliseconds. This reflects from the number of nodes that were explored, greedy best first search searched the least number of nodes to find the goal while Dijkstra was one of the highest. Breadth-first-search, Dijkstra, A*, Min-heap A* we're all able to produce optimal paths with certain heuristics. Depth-first-search, greedy best first search and min heap greedy best first search produced un-optimal paths. Optimal and non-optimal path behaviour for each algorithm refer to **Appendix D**.

There is an issue with the min heap optimal map. This is caused when two nodes equal the same amount, but one represents a straight direction while the other represents a diagonal direction. As mentioned previously straight directions are prioritised first so when the min heap takes from the bottom of the array to the top then the checks are done to move it down the array if the value of the node equals the same then it doesn't move and stays at the top. This results in a node that represents a diagonal direction being picked instead of a straight direction node. **Appendix G**

5.5 - Map Two Maze

5.5.1 - Table [7]: Time it takes to find a path.

Map Two - Time it takes to find a path	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	1.69
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	0.64
Dijkstra	1.69	1.70	1.68	1.67	1.64	N/A
Greedy-Best-First-Search	0.79	0.78	0.85	0.58	0.79	N/A
A* Pathfinding	1.71	1.64	1.67	1.70	1.67	N/A
MinHeap - Dijkstra	1.72	1.70	1.70	1.71	1.71	N/A
MinHeap - Greedy-Best-First-Search	0.84	0.78	0.85	0.61	0.79	N/A
MinHeap - A* Pathfinding	1.71	1.73	1.76	1.74	1.67	N/A

5.5.2 - Table [8]: Number of Nodes explored.

Map Two - Nodes Explored	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	401
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	228
Dijkstra	401	401	401	401	401	N/A
Greedy-Best-First-Search	256	250	256	214	256	N/A
A* Pathfinding	398	400	398	398	398	N/A
MinHeap - Dijkstra	401	401	401	401	401	N/A
MinHeap - Greedy-Best-First-Search	256	250	256	214	256	N/A
MinHeap - A* Pathfinding	398	397	398	398	398	N/A

5.5.3 - Table [9]: Number of nodes in path to goal.

Map Two - Nodes to Goal/Path	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	109
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	110
Dijkstra	109	109	109	109	109	N/A
Greedy-Best-First-Search	110	110	110	109	110	N/A
A* Pathfinding	109	109	109	109	109	N/A
MinHeap - Dijkstra	109	109	109	109	109	N/A
MinHeap - Greedy-Best-First-Search	110	110	110	109	110	N/A
MinHeap - A* Pathfinding	109	109	109	109	109	N/A

5.5.4 - Analysis of Result.

Overall, the slowest time is min heap A* using the Euclidean heuristic taking the average time of 1.76 milliseconds each cycle. The fastest time is the best first search using the Chebyshev heuristic taking 0.58 milliseconds. Dijkstra and breadth-first-search explored the most nodes most the time it was faster than A*. In this scenario A* is unable to show its full benefit so the extra calculations A* does make it more expensive with little gain. greedy best first search searched the least number of nodes again resulting in a faster time. Optimal and non-optimal path behaviour for each algorithm refer to **Appendix E**.

What was unexpected was that all the weighted algorithms were able to produce the most optimal path when using the Chebyshev distance, this could be due to the size of the map and the size of the corridors. Depth-first-search was able to display its potential although unable to find the most optimal path it was able to find the goal in half the time of breadth-first-search.

5.6 - Map Three Terrain

5.6.1 - Table [10]: Time it takes to find a path.

Map Three - Time it takes to find a path	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	4.07
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	6.63
Dijkstra	16.28	16.35	16.41	15.72	16.32	N/A
Greedy-Best-First-Search	1.56	0.49	1.60	1.86	1.60	N/A
A* Pathfinding	7.52	7.22	8.62	7.97	7.48	N/A
MinHeap - Dijkstra	14.82	14.93	14.87	13.96	14.77	N/A
MinHeap - Greedy-Best-First-Search	0.95	0.29	1.02	0.97	0.97	N/A
MinHeap - A* Pathfinding	6.13	5.16	7.31	6.25	6.15	N/A

5.6.2 - Table [11]: Number of Nodes explored.

Map Three - Nodes Explored	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	414
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	410
Dijkstra	843	847	844	828	843	N/A
Greedy-Best-First-Search	187	88	182	205	187	N/A
A* Pathfinding	536	525	583	556	536	N/A
MinHeap - Dijkstra	841	851	842	821	841	N/A
MinHeap - Greedy-Best-First-Search	186	89	182	189	186	N/A
MinHeap - A* Pathfinding	534	490	583	540	534	N/A

5.6.3 - Table [12]: Number of nodes in path to goal.

Map Three - Nodes to Goal/Path	Diagnol	Manhatten	Euclidean	Chebyshev	Octile	No-Heuristic
Breadth-First-Search	N/A	N/A	N/A	N/A	N/A	20
Depth-First-Search	N/A	N/A	N/A	N/A	N/A	396
Dijkstra	38	38	38	38	38	N/A
Greedy-Best-First-Search	41	41	41	41	41	N/A
A* Pathfinding	38	38	38	38	38	N/A
MinHeap - Dijkstra	38	38	38	38	38	N/A
MinHeap - Greedy-Best-First-Search	41	41	41	42	41	N/A
MinHeap - A* Pathfinding	38	39	38	38	38	N/A

5.6.4 - Analysis of Result.

Dijkstra using the Euclidean heuristic is the slowest algorithm taking 16.41 milliseconds while the min heap best-first-search with Manhattan heuristic is the fastest algorithm talking 0.29 milliseconds. Both Dijkstra implementations produce the highest number of nodes explored all being above 800. Greedy best first search algorithm searched the least number of nodes.

Dijkstra, A*, Minheap and min heap A* was able to produce the most optimal routes with all heuristics except min heap A*. Greedy best first search producing the most un-optimized path. Optimal and non-optimal path behaviour for each algorithm refer to **Appendix F**.

Breadth-first-search and depth-first-search are unable to produce accurate results on this map due to the nodes having weight on them. The algorithms are unable to see how much a node is worth, hence although it seems like breadth-first-search has the best path when you add the values all together it will have a less optimal path.

5.7 - Conclusion of Results

After analysing all the data, the results show each pathfinding algorithm is suitable for different purposes and the scenario that it is being used for. If you are wanting the fastest path without any concern for the optimal path, then you would want to use best-first-search. If you are wanting the most optimal path but aren't concerned about the speed, then you would use Dijkstra. If wanting the fastest time while producing the most optimal path, then you would use A*.

If you are wanting to use non-weighted algorithms then you would use breadth-first-search the only time you would use depth-first-search is when you are wanting to use it to find the path through a maze where it is the fastest. You must consider the maze corridor size if the width and length exceed two then you will have the same issues that you see in the other maps.

When each test was run for the heuristics it was completely different each cycle, this meant that it was stochastic making it impossible to determine which heuristic was the fastest due to the inconsistency of the results.

What can be confirmed though from looking at the data is that if you are looking for the heuristic that provides the most consistent optimal paths then using octile, diagonal or Euclidean would be the best option.

The Minheap did show speed improvements the most noticeable was seen in map three. Map one did show speed improvements but was miniscule, while map two the outcome was the min heap was much slower. Using the results gathered from the test the min should be applied on much larger maps and weighted maps to be able to get the full benefits from it.

"Dijkstra's algorithm, breadth first search algorithm and depth first search algorithm, were created to solve the shortest path problem until the emergence of A* algorithm as a provably optimal solution for pathfinding." - *(Cui. X & Shi. H, 2011).* This statement can be shown to be true in most cases as seen in map one and map three, but the data shown in map two shows that A* is not always the most optimal solution compared to the other algorithms.

The Flowfield algorithm is the most efficient in terms of handling large number of units, no test was created to display this. However, when instantiating units using single-pathfinding algorithms in the application the computer started to struggle dropping frame rate and freezing. This is down to each unit having to have a unique path calculated for it. On the other hand, Flowfield calculates the grid once and the unit only needs to look at the grid to get the direction. This was demonstrated when instantiating the same number of units but the frame rate staying the same. It also made it easier to implement unit collision avoidance compared to single-agent pathfinding.

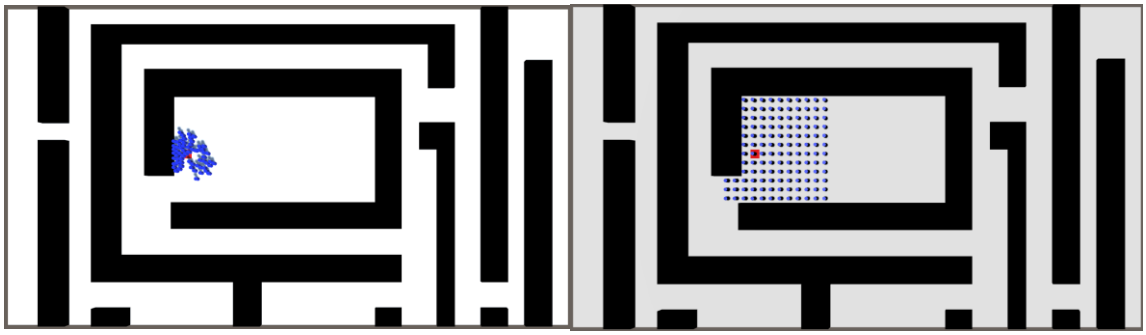


Figure [27]: Left single agent pathfinding and right multi-agent pathfinding

6 - Critical Reflection & Evaluation

This section will look back at the development of the project from start to finish and reflect on how successful the project went, looking back to see what could have been done differently to improve the overall deliverable.

6.1 - Critique of Research

Many aspects of the project were taken into consideration when delving into the research. Sufficient research was gathered, providing us the techniques and methods to successfully achieve the aims of the deliverable. The original goal of the project was to apply a basic and advanced algorithm then compare them. However, when analysing the research and going through numerous material it was possible to implement more algorithms. Ultimately changing the scope and aims of the project.

Various heuristic methods were researched and considered for their application to the aim of providing path optimality and speed improvements. The initial candidate for satisfying this goal was Octile Distance alone. In the end, the project pursued multiple algorithms to give the user more options to choose from and demonstrating the effect on the algorithms. Improving the overall user experience.

One topic that should have been researched and investigated was the Big-O-Notation terminology. These were used in several research papers and provided the information on how long an algorithm takes to run. This could have allowed me to save time in certain areas avoiding using certain processes or functions. One example is using the lists functionality in the min heap. The terminology could have also been used in the testing section when demonstrating my results in a more scientific way.

After project completion, further research should investigate in depth, the advanced algorithms like D* lite, LPA* and IDA* analysing what separates them from each other and applying them in future work. When moving onto future projects more focus would be spent on the main research aims, making sure enough research is provided before moving onto the less important aims of the project.

6.2 - Critique of Design

The design is the one I felt was the least developed out of all the sections, with the pressure of entering the unknown and having no prior knowledge of the subject. It made me decide to develop my deliverable from a basic tutorial, due to the uncertainty of being unable to deliver the project aims. The gains from this is that I didn't have to concern myself with the architecture and class design since they were already paved out for me. It provided me the opportunity to work on the more crucial points of my project like the algorithms, dynamic map, heuristics, user interface and unit movement. The issue though is that it made me miss out on a huge opportunity on choosing the architecture and design of the deliverable itself. It made it difficult to see what functions should be where and if new classes should be created for certain features. Going forward in future work, more investigation needs to be met when designing the architecture and design of a project.

With the amount of documentation and research that can be found, the information provided granted the opportunity to create a pseudocode design for each algorithm. This provided a better insight into how each algorithm was to be implemented and how the algorithms work. When it came to develop the Flowfield, it proved the knowledge gained prior allowed me a better grasp on how

the Flowfield worked, making it smoother to implement. It also provided the reader to see into the thought process the author was thinking at the time.

The User interface was originally designed to be rudimentary with limited functionality allowing the user only the basic control. It was only when working through the development that the importance of a more user-friendly interface was made clear and decided to expand more on it. Creating a visual aid toggle, allowing the user a choice of inputting where they want the goal, inputting the location of where they want the unit and creating a console log to display information from the algorithms from each unit. This not only made it easier for testing but allowed the user more control and a better experience.

The biggest issue in this whole project was the minimum binary heap just because a concept works doesn't mean that the design implemented would achieve the intended purpose. This was found out the hard way, when the min heap was created using lists it didn't produce the expected results this meant going back and looking for other ways to implement the minimum binary heap. This could have been avoided if alternative designs were investigated instead of settling on the concept alone. In future work, more investigation into different implementation alternatives will be looked into instead of settling on the first design.

6.3 - Development & future work

The finalized deliverable can demonstrate various algorithms with applied optimization techniques. With a visual representation and information being displayed to see how each one differs. Overtime this application can be developed into a learning tool for students to be able to understand the intricacy of Artificial intelligence pathfinding. The final deliverable met every project aim from the project specification, however, with hindsight more efficient functionality and features could have been implemented to improve the overall deliverable.

Although the deliverable met all its target, issues arose in certain scenarios which in hindsight could have been avoided. If redoing the application with the skills gained from this project. Instead of using the Unity game engine, it would be created in a C++ standalone application. When testing the results in Unity, not being able to understand what processes are going on in the background is a huge problem. This affected the results of the testing making it unreliable when trying to get accurate results for the heuristics times. With a C++ standalone application, it means building it from the ground up, allowing for more control of what is going on and more efficient optimization. Before attempting this future endeavour, more investigation and research would need to be taken into consideration when looking into the architecture and class design.

One of the biggest regrets was not being able to implement Hierarchical pathfinding into the project. When developing the dynamic map/pathfinding it was realised then, how powerful this could have been in terms of optimization. Take Flowfield every time a change has been made to the terrain it recalculates the whole map, with hierarchical pathfinding it would only change the section of the map. It can also be applied for when repainting the map. The result of applying this implementation would result in a much-improved performance and efficiency.

Further development for the unit movement would need to be done for the basic algorithms when multiple units are on the map due to them getting stuck, but this is down to the limitations of the code and algorithms. If changes were made to the algorithms would it still be called A* or would it be a totally different algorithm entirely? This would need further investigation to find the answer.

The Flowfield demonstrated efficient movement and optimization when handling large number of units, it's my favourite algorithm out of the whole project. There will need to be improvements made to the unit movement to avoid units getting sandwiched and being trapped. Looking into how to improve the algorithm when used on terrain types maps and although it worked, the outcome wasn't what I fully intended for it to do. Once these are done the next step is being able to create multiple groups of units getting them to travel to different destinations on the map while being able to avoid other groups.

The original implementation of the minimum binary heap relied on the Lists built in functionality, this was incorporated into the min heaps functions. This resulted in it being slower and didn't produce the results I had expected. Moving onto a new implementation that used an array which resulted in improvement within the speed, but another issue rose with the optimal paths mentioned in both **section 4.6 and 5.4.4**. The solution to resolve this could have been when the node neighbours' values are being calculated, add onto it a miniscule incremented amount to the diagonal nodes to prevent the values being the same. This wasn't tested due to a strict rule to not touch the deliverable once past the deadline unless it was minor changes. When carrying on developing this project in the future, the minimum binary heap is the one that would need developing further.

The deadline dates set out for each target were able to be met without any hassle, making sure deadlines for the project were completed a few days before the date. No project management tools needed to be relied on, but it would have been nice to see a clear indication of the progress of where the project was at. Next time a project management tool will be used to provide a visual representation of the progress of the project.

6.5 - Critique of Testing

The testing achieved what it was set out to do, showing the difference between each algorithm in terms of speed and path optimality. During the testing, it was realised two more maps would need to be created. This was to demonstrate the effectiveness of depth-first-search and displaying the difference between weighted and unweighted algorithms since the first map was unable to do this unless the user altered the terrain.

Once each algorithm was tested, a table was implemented to represent the data to the user allowing them to pinpoint the more informative information. The table was colour coded showing the best results green and the worst results red. The orange is a special case, that was to be spoken about in the analyse section of the report.

Unfortunately, due to the limitations of using Unity it wasn't possible to determine the fastest heuristic due to the results being completely different each run. Although it made determining the heuristics near impossible it did demonstrate that it was stochastic.

An opportunity missed was implementing a frame rate test between the Flowfield algorithm and the basic algorithms. The test would involve incrementing the amounts of units within the scene till it's reached a certain number of frames. Then this would have provided clear evidence to say that the Flowfield is more efficient when handling larger amounts of units. This idea only came about at the end of the project hand-in which is why it has not been implemented or mentioned. If any future testing is to be done, then this will be the first one that will be conducted.

6.6 - Personal & Professional Learning

The development of the project had made me realise the potential it has to be used as a learning application for students and teachers. Developed as an open source application, it would be able to guide the younger generation to be more involved in artificial intelligence. Of course, more work would have to be done to reach this goal.

Upgrading the graphics of the map and models making it more detailed to be able to achieve the aims of captivating the user and keeping them interested. Applying a much better designed user interface with more algorithm options to choose from. Implementing the hierarchical pathfinding improving the performance and allowing the user to be able to move a group of units to different locations. Sifting through these features prioritising the ones with more importance to be developed. Using the knowledge & skills learnt from this project and applying them when making these future developments.

In the development of this project I have managed to overcome the pressure and uncertainty that I originally had at the start. The pressure allowed me to prioritise more important tasks allowing me to better manage the way the project went, allocating time where it was necessary.

Like a beacon this project has opened my eyes to the direction I would like to take myself in the industry. I've only managed to touch the tip of the iceberg but have gained valuable knowledge from this. Allowing me to better understand the different aspects of AI algorithms and their intricacies that has allowed me to develop the skills required to get a foothold in the AI industry.

To advance my skills further I'll be looking to apply myself to multi-agent algorithms designed for security simulations. Looking at applying for a job in the security industry and advancing my skills learning the industry standards of AI crowd simulations, pushing myself further down the road of AI. Although trying at times I've had the greatest pleasure working on this topic, if I apply the same amount of effort and enthusiasm as on this project then the future looks bright.

References

A* Pathfinding Project: Graph Types. (2020). Retrieved 3 April 2020, from https://arongranberg.com/astar/documentation/3_8_11_16fd74a1/graph_types.php

Abd Algfoor, Z., Sunar, M. S., & Kolivand, H. (2015). A comprehensive study on pathfinding techniques for robotics and video games. International Journal of Computer Games Technology, 2015.

Amazon Lumberyard: Features. (2020). Retrieved 15 March 2020, from https://aws.amazon.com/lumberyard/details/

Amit Patel Heuristics. (2020). Retrieved 15 March 2020, from http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Amit Patel Map representations. (2020). Retrieved 15 March 2020, from http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html

Anderson, R. (2020). Stacks and Queues. Retrieved 3 April 2020, from https://everythingcomputerscience.com/discrete_mathematics/Stacks_and_Queues.html

B. Sobota, C. Szabo and J. Perhac, "Using path-finding algorithms of graph theory for route-searching in geographical information systems," 2008 6th International Symposium on Intelligent Systems and Informatics, Subotica, 2008, pp. 1-6

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. Journal of game development, 1(1), 7-28.

Breadth First Search Tutorials & Notes | Algorithms | HackerEarth. (2020). Retrieved 15 March 2020, from https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

CRYENGINE | Features. (2020). Retrieved 15 March 2020, from https://www.cryengine.com/features

Cui. X & Shi. H, (2011). A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security, 11(1), 125-130.

Depth First Search Tutorials & Notes | Algorithms | HackerEarth. (2020). Retrieved 15 March 2020, from https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Duc, L. M., Sidhu, A. S., & Chaudhari, N. S. (2008). Hierarchical pathfinding and ai-based learning approach in strategy game design. International Journal of Computer Games Technology, 2008.

Durant. S, (2013). Understanding Goal-Based Vector Field Pathfinding. Retrieved 15 March 2020, from https://gamedevelopment.tutsplus.com/tutorials/understanding-goal-based-vector-field-pathfinding--gamedev-9007

Edsger W. Dijkstra - A.M. Turing Award Laureate. (2020). Retrieved 15 March 2020, from https://amturing.acm.org/award_winners/dijkstra_1053701.cfm

Emerson. E, (2013). Crowd pathfinding and steering using flow field tiles. Game AI Pro: Collected Wisdom of Game AI Professionals, 307-316.

Euclidean vs Chebyshev vs Manhattan Distance. (2012). Retrieved 15 March 2020, from https://lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/

Felner, A. (2011, July). Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In Fourth annual symposium on combinatorial search.

Flow Field Pathfinding. (2020). Retrieved 15 March 2020, from https://leifnode.com/2013/12/flow-field-pathfinding/

Graham. Ross; McCabe. Hugh; and Sheridan. Stephen (2003) "Pathfinding in Computer Games," The ITB Journal: Vol. 4: Iss. 2, Article 6.

Graph Theory — Basic Properties. (2019). Retrieved 3 April 2020, from https://towardsdatascience.com/graph-theory-basic-properties-955fe2f61914

Grids and Graphs. (2020). Retrieved 15 March 2020, from https://www.redblobgames.com/pathfinding/grids/graphs.html

Grishechko, E. (2018). Max and Min heap implementation with C#. Retrieved 15 March 2020, from https://egorikas.com/max-and-min-heap-implementation-with-csharp/

Introduction to Priority Queues using Binary Heaps - Techie Delight. (2016). Retrieved 15 March 2020, from https://www.techiedelight.com/introduction-priority-queues-using-binary-heaps/

Kim. H, Yu. K, & Kim. J, (2011). Reducing the Search Space for Pathfinding in Navigation Meshes by Using Visibility Tests. Journal of Electrical Engineering & Technology, 6(6), 867-873.

Kuffner, J. J. (2004). Efficient optimal search of uniform-cost grids and lattices. In 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566) (Vol. 2, pp. 1946-1951). IEEE.

Kumar, P., Bottaci, L., Mehdi, Q., Gough, N., & Natkin, S. (2004). Efficient path finding for 2D games.

Lague, Sebastian. (2014). A* pathfinding (E04:heap optimization) [VIDEO]. Retrieved 15 March 2020, from https://www.youtube.com/watch?v=3Dw5d7PlcTM&t=492s

Mariam Arshad, Arwa Al-Issa, S. S. Z. H. A.-J. , M. B. , L. A.-J. ,. (2017). Parallelizing A* Path Finding Algorithm. International Journal of Engineering and Computer Science, 6(9). Retrieved 15 March 2020, http://www.ijecs.in/index.php/ijecs/article/view/2774

Mehta. Parth & Shah. Hetasha & Shukla, Soumya & Verma, Saurav. (2015). A Review on Algorithms for Pathfinding in Computer Games.

Nagelkerke. T, (2016). Navigation to a human in motion by using points of interest.

N. Sturtevant. “Choosing a search space representation.” In Game AI Pro, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.

Norvig. P R, & Intelligence, S. A. (2002). A modern approach. Prentice Hall. P 92 – p93

Overview of Visual Studio. (2019). Retrieved 15 March 2020, from https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019

Pandit. S & Gupta. S, (2011). A comparative study on distance measuring approaches for clustering. International Journal of Research in Computer Science, 2(1), 29-31.

Patrick Lester, Heuristics and A* Pathfinding. (2020). Retrieved 15 March 2020, from https://web.archive.org/web/20171019182159/http://www.policyalmanac.org/games/heuristics.htm

Rabin. S, & Sturtevant. N. R, (2013). Pathfinding architecture optimizations. Game AI Pro: Collected Wisdom of Game AI Professionals, 1, 241-252.

Satre Meloy. A, Diakonova. M, & Grunewald, P. (2019, June). What makes you peak? Cluster analysis of household activities and electricity demand. European Council for an Energy Efficient Economy.

Technologies, U. (2020). Unity - Manual: Unity User Manual (2019.3). Retrieved 15 March 2020, from https://docs.unity3d.com/Manual/UnityMan

Unreal Engine | Features. (2020). Retrieved 15 March 2020, from https://www.unrealengine.com/en-US/features

Wilmer, Lin. Learn A* & Graph Search Algorithms in Unity: Pathfinding in Unity. (2020). Retrieved 15 March 2020, from https://www.udemy.com/course/pathfinding-in-unity/

Zhang, A., Li, C., & Bi, W. (2016). Rectangle expansion A∗ pathfinding for grid maps. Chinese Journal of Aeronautics, 29(5), 1385-1396.

Appendix Summary

Appendix A - Algorithm path behaviour.

This section shows the algorithms path behaviour. The grey represents the nodes that have already been expanded, the purple are the nodes that are waiting to be expanded and the blue is the path.

A.1 - Breadth-First-Search

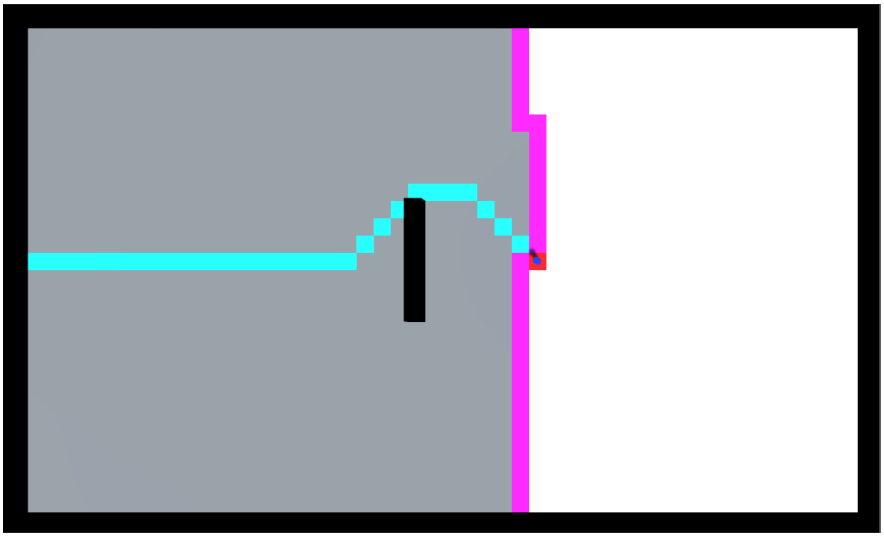


Figure [28]: Breadth-First-Search path behaviour.

A.2 - Depth-First-Search

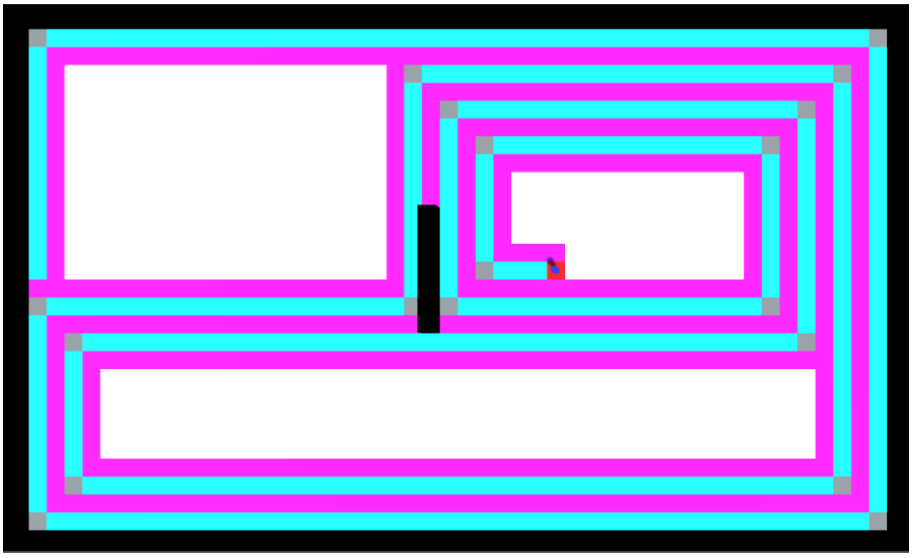
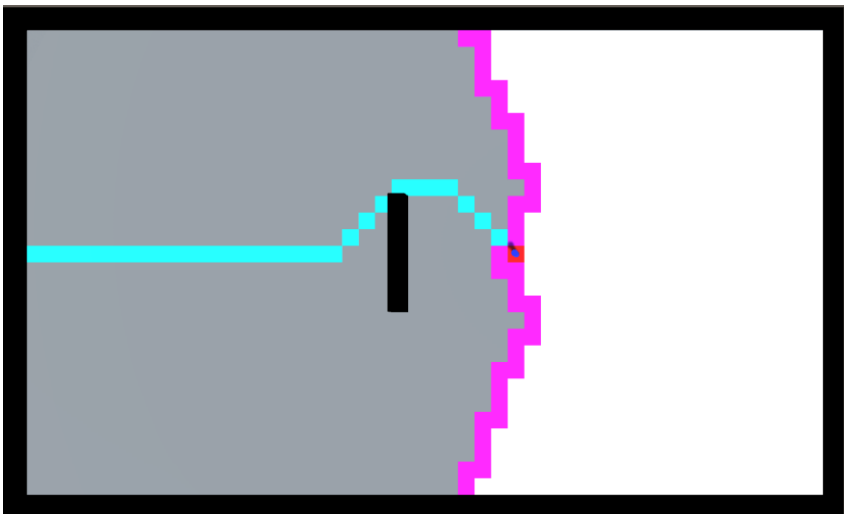


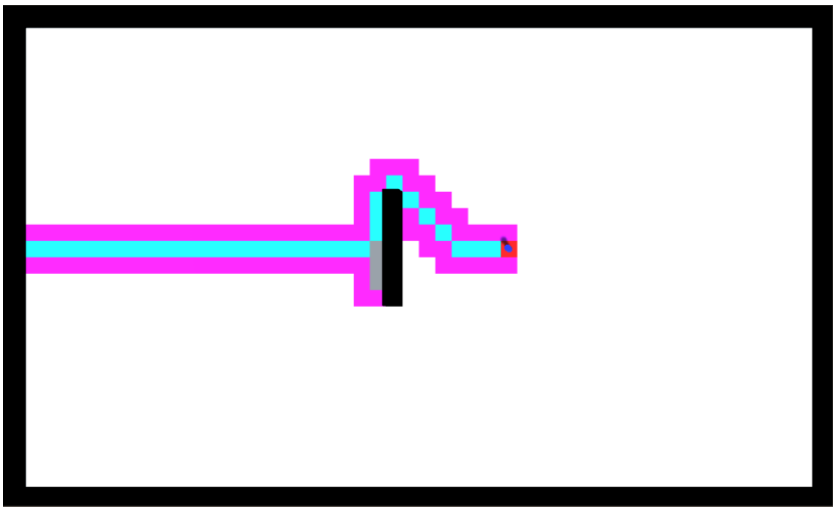
Figure [29]: Depth-First-Search path behaviour.

A.3 - Dijkstra Variant Uniform Cost Search



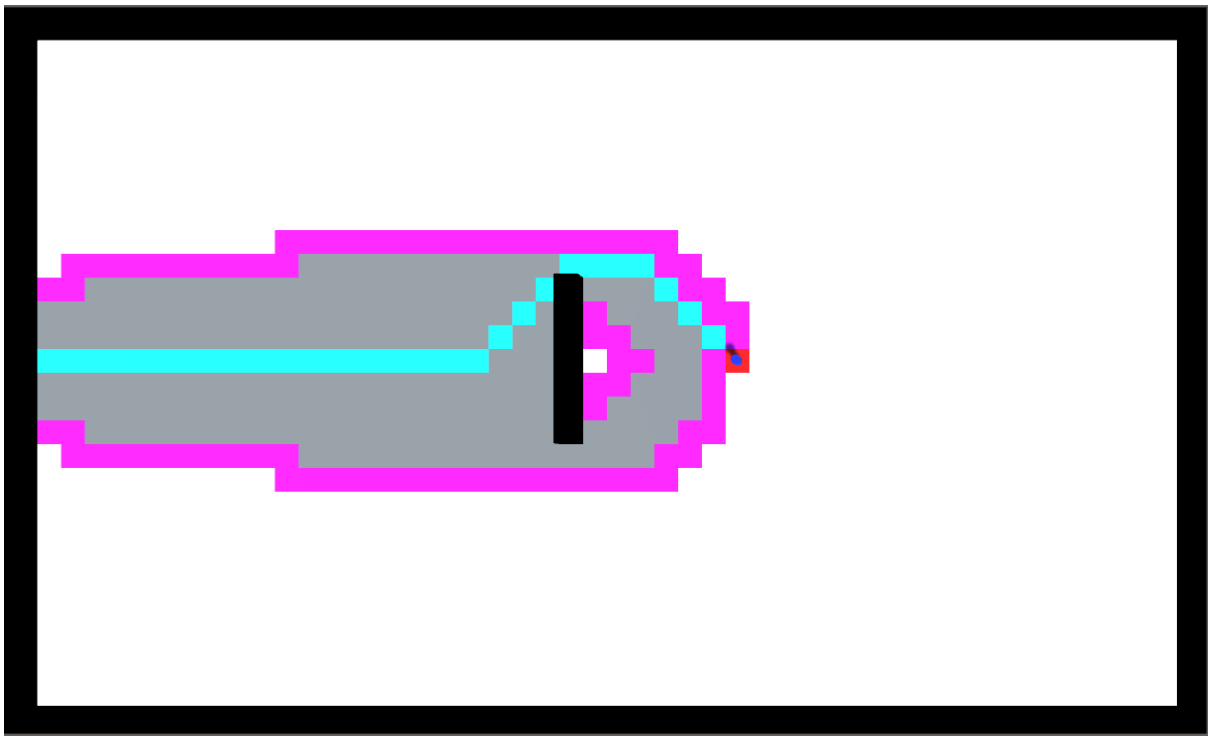
Figure[30]: Dijkstra variant, uniform cost search path behaviour.

A.4 - Greedy-Best-First-Search

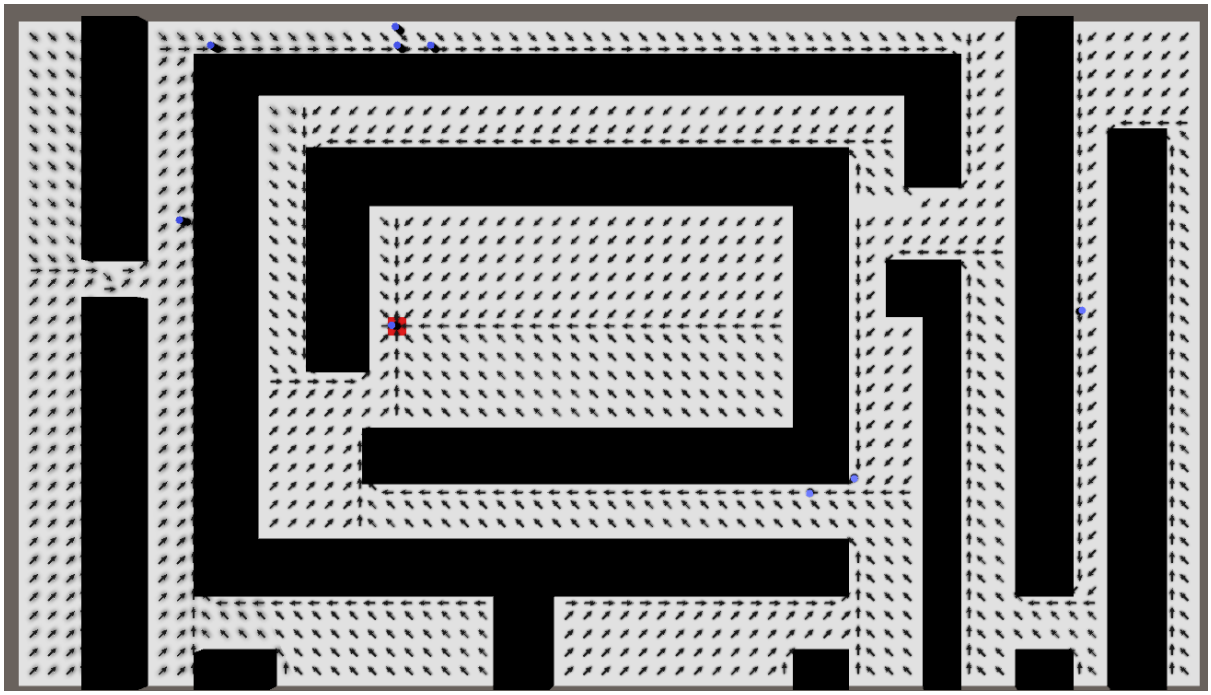


Figure[31]: Greedy-Best-First-Search path behaviour.

A.5 - A*



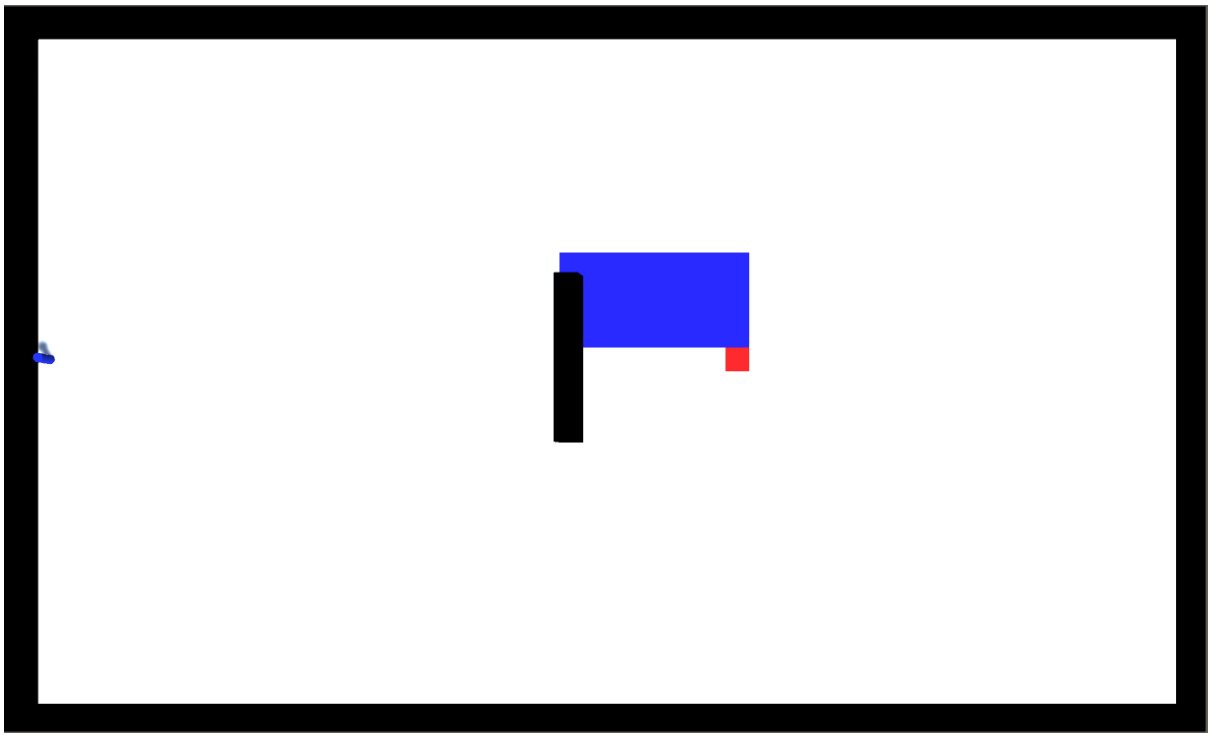
Figure[32]: A path behaviour.*



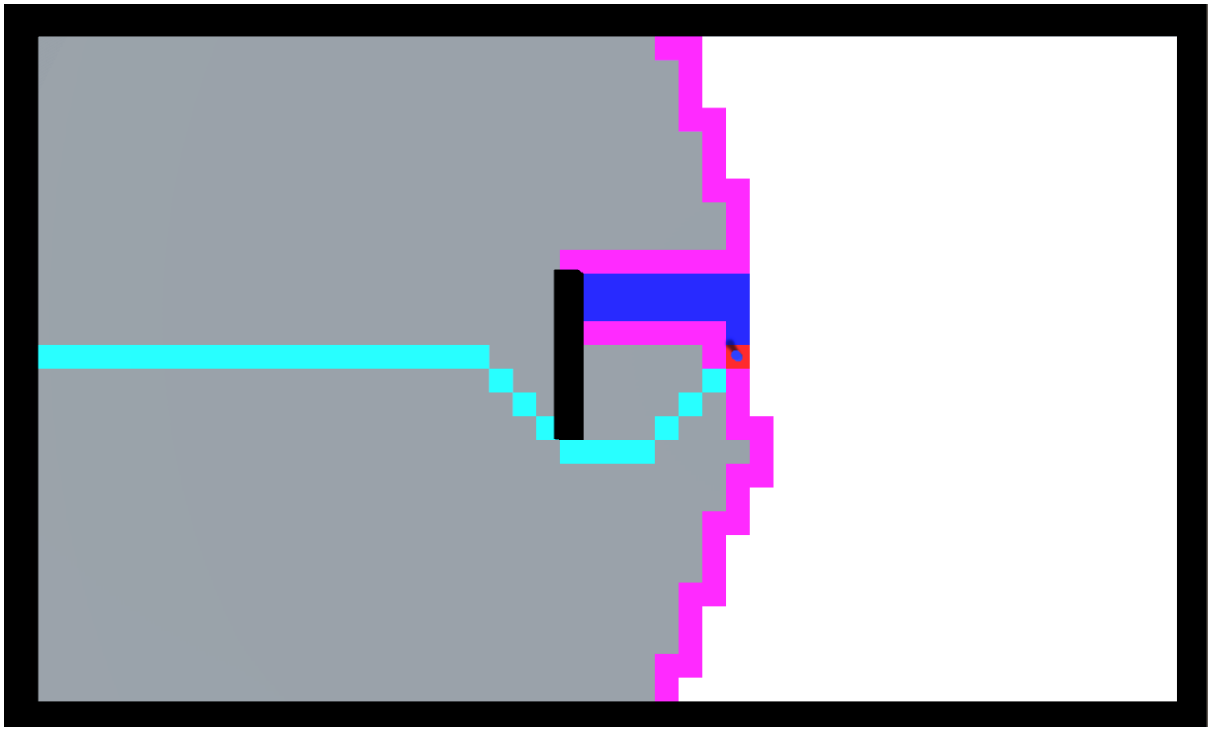
Figure[33]: Flowfield Algorithm flow/path behaviour.

A.6 - Weighted and Non-Weighted Algorithm Behaviour

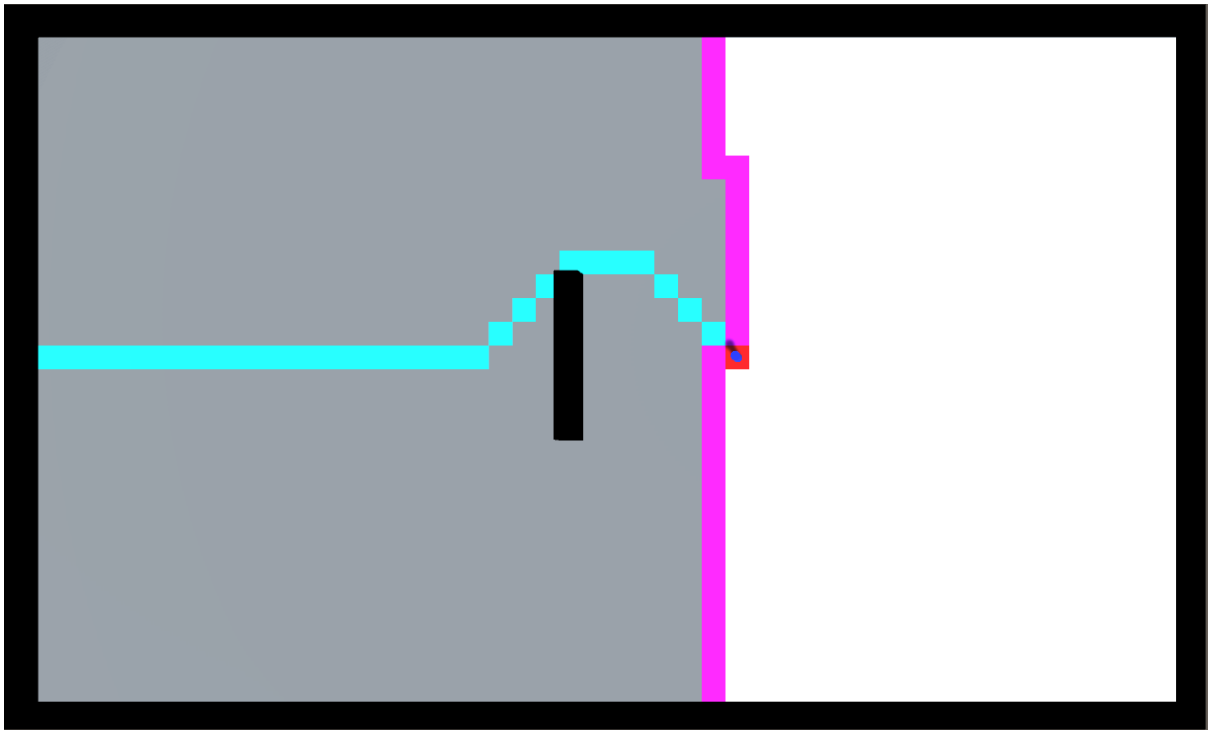
Non-Weighted algorithms will ignore the cost of the terrain while weighted algorithms will take the cost into consideration.



Figure[34]: Adding weighted value to nodes, blue represents sea with a high cost to cross.



Figure[35]: Weighted Algorithm Dijkstra path behaviour.



Figure[36]: Non-weighted Algorithm Breadth-First-search path behaviour.

Appendix B - Pseudocode Pathfinding Algorithms

Appendix B holds the pseudocode that I will follow to help develop each algorithm for my project.

B.1 - Breadth-First-Search Pseudocode

```
BFS (s, g):                 //s is the start node, g is the goal node.
    let OL be list.         // Represents our list of nodes to expand.
    let CL be list.         // Represents our list of nodes explored.
    let PL be list          // Represents our list of nodes to the goal.

    OL.Add( s )             //Add s into the O list.

    while ( OL is not empty)
            n = OL[0]                //Get first node in OL and store in n
            OL.Remove( n )          //Remove n from OL

            if n is not in CL
                CL.Add( n )         //Add n into CL

            //process the neighbour list that will be in n
            for all nodes w in n.neighbour
                if w is not in CL && w is not blocked && w is not in OL
                        w.parent = n            //n is parent node of w used for retracing path
                        OL.Add( w )             //Stores w in OL to further visit its neighbours

            If g is in OL
                PL = retrace path( g )     //Function to retrace path using the goal node
                Return PL                  //Return the path list back

  Return unable to find path        // goal has not been found, can't find path
```

B.2 - Depth-First-Search Pseudocode

```
DFS (s, g):                 //s is the start node, g is the goal node.
    let OL be list.         // Represents our list of nodes to expand.
    let CL be list.         // Represents our list of nodes explored.
    let PL be list          // Represents our list of nodes to the goal.

    OL.Add( s )             //Add s into the O list.

    while ( OL is not empty)
            n = OL.Last()           //Get last node in OL and store in n (difference between BFS)
            OL.Remove( n )          //Remove n from OL

            if n is not in CL
                CL.Add( n )         //Add n into CL

            //Process the neighbour list that will be in n
            for all nodes w in n.neighbour
                if w is not in CL && w is not blocked && w is not in OL
                        w.parent = n            //n is parent node of w used for retracing path
                        OL.Add( w )             //Stores w in OL to further visit its neighbours

            If g is in OL
                PL = retrace path( g )      //Function to retrace path using the goal node
                Return PL                   //Return the path list back

  Return unable to find path        // Goal has not been found, can't find path
```

B.3 – Dijkstra Pseudocode

```
Dijkstra (s, g, grid):                          //s is the start node, g is the goal node, grid is the grid of nodes
    for each node n in grid                     //reset values of all nodes in grid
            n.gCost = INFINITY
            n.parent = UNDEFINED

    let OL be list.         // Represents our list of nodes to expand.
    let CL be list.         // Represents our list of nodes explored.
    let PL be list          // Represents our list of nodes to the goal.

    s.gCost = 0             //set s.gCost to zero.
    OL.Add( s )             //Add s into the O list.

    while ( OL is not empty)
            n =  node in OL with minimum n.gCost        //Get node with lowest gCost in OL
            OL.Remove( n )          //Remove n from OL

            if n is not in CL
                CL.Add( n )         //Add n into CL

            //process the neighbour list that will be in n
            for all nodes w in n.neighbour
                if w is in CL || w is blocked
                        continue move onto next w       //skip everything below move onto next w

                distancecost = n.gCost + GetDistance(n, w)  //add node cost with distance between n and w

                if distancecost is less than w.gCost || w is not in OL
                        w.gCost = distancecost              //w.gCost becomes distancecost
                        w.parent = n                        //n is parent node of w used for retracing path

                        if w is not in OL
                            OL.Add( w )                     //Stores w in OL to further visit its neighbours

            If g is in OL
                PL = retrace path( g )      //Function to retrace path using the goal node
                return PL                   //Return the path list back

  return unable to find path        // Goal has not been found, can't find path
```

B.4 - Greedy-Best-First-search Pseudocode

```
Greedy best-first search  (s, g, grid):          //s is the start node, g is the goal node, grid is the grid of nodes
    for each node n in grid                      //reset values of all nodes in grid
            n.hCost = INFINITY
            n.parent = UNDEFINED

    let OL be list.        // Represents our list of nodes to expand.
    let CL be list.        // Represents our list of nodes explored.
    let PL be list         // Represents our list of nodes to the goal.

    s.hCost = GetDistance(s, g)  //estimate heuristic cost from s to g
    OL.Add( s )            //Add s into the O list.

    while ( OL is not empty)
            n =  node in OL with minimum n.fCost          //Get node with lowest n.fCost in OL
            OL.Remove( n )          //Remove n from OL

            if n is not in CL
                 CL.Add( n )        //Add n into CL

            //process the neighbour list that will be in n
            for all nodes w in n.neighbour
                if w is not in CL && w is not blocked && w is not in OL
                        heuristicestimate =  Getdistance(w, g)          //estimate the distance between w and g
                        w.hCost = heuristicestimate
                        w.parent = n                          //n is parent node of w used for retracing path

             If g is in OL
                PL = retrace path( g )     //Function to retrace path using the goal node
                return PL                  //Return the path list back

  return unable to find path        // goal has not been found, can't find path
```

B.5 - A* Pseudocode

```
A* (s, g, grid):                        //s is the start node, g is the goal node, grid is the grid of nodes
    for each node n in grid                     //reset values of all nodes in grid
            n.gCost = INFINITY
            n.hCost = INFINITY
            n.parent = UNDEFINED

    let OL be list.         // Represents our list of nodes to expand.
    let CL be list.         // Represents our list of nodes explored.
    let PL be list          // Represents our list of nodes to the goal.

    s.gCost = 0             //set s.gCost to zero.
    s.hCost = GetDistance(s, g)  //estimate heuristic cost from s to g
    OL.Add( s )             //Add s into the O list.

    while ( OL is not empty)
            n =  node in OL with minimum n.fCost          //Get node with lowest n.fCost in OL
            OL.Remove( n )          //Remove n from OL

            if n is not in CL
                CL.Add( n )         //Add n into CL

            //process the neighbour list that will be in n
            for all nodes w in n.neighbour
                if w is in CL || w is blocked
                        continue move onto next w         //skip everything below move onto next w

                distancecost = n.gCost + GetDistance(n, w)   //add node cost with distance between n and w
                heuristicestimate =  Getdistance(w, g)        //estimate the distance between w and g

                if distancecost is less than w.gCost || w is not in OL
                        w.gCost = distancecost
                        w.hCost = heuristicestimate
                        w.fCost = distancecost + heuristicestimate  //Total cost for the node
                        w.parent = n                          //n is parent node of w used for retracing path

                        if w is not in OL
                            OL.Add( w )                       //Stores w in OL to further visit its neighbours

            If g is in OL
                PL = retrace path( g )     //Function to retrace path using the goal node
                return PL                  //Return the path list back

  return unable to find path        // goal has not been found, can't find path
```

B.6 - Flowfield Pseudocode

```
Cost field (g, grid):                          //g is the goal node, grid is the grid of nodes
    for each node n in grid                    //reset values of all nodes in grid
            n.gCost = INFINITY
            n.parent = UNDEFINED

    let OL be list.        // Represents our list of nodes to expand.
    let CL be list.        // Represents our list of nodes explored.

    s.gCost = 0            //set s.gCost to zero.
    OL.Add( g )            //Add g into the O list.

    while ( OL is not empty)
            n =  node in OL with minimum n.gCost        //Get node with lowest gCost in OL
            OL.Remove( n )          //Remove n from OL

            if n is not in CL
                CL.Add( n )         //Add n into CL

            //process the neighbour list that will be in n
            for all nodes w in n.neighbour
                if w is in CL || w is blocked
                        continue move onto next w       //skip everything below move onto next w

                distancecost = n.gCost + GetDistance(n, w)  //add node cost with distance between n and w

                if distancecost is less than w.gCost || w is not in OL
                        w.gCost = distancecost              //w.gCost becomes distancecost
                        w.parent = n                        //n is parent node of w used for retracing path

                        if w is not in OL
                            OL.Add( w )                     //Stores w in OL to further visit its neighbours

            grid.IntegrationField   //When OL is empty call integration field
```

```
Integration field ():                  //grid is global in this class
    for each node n in grid                    //cycle through all nodes in grid

        if n is blocked
            continue move onto next n

        n.parent = n     // Make n the parent node.

        for all nodes w in n.neighbour
            if w is blocked
               continue move onto next w

             if w.gCost is less than n.parent.gCost  //if w gCost is lower replace the node parent
                  n.parent = w
```

Appendix C - Code Implementations

This appendix holds all the code snippets used in the specific parts of the project.

C.1 - Map Generation & Map design

```
void Start()
{
    if (mapData != null && grid != null)
    {
        int[,] mapDesign = mapData.MakeMap();
        grid.CreateGrid(mapDesign);

        gridVisualisation = grid.gameObject.GetComponent<GridVisualisation>();

        if (gridVisualisation != null)
        {
            gridVisualisation.CreateGridVisualisation(grid);
        }
```

Figure [37]: Start function in controller classes.

```
public int[,] MakeMap() //2d array
{

    int[,] map;
    if (textureMap != null)
    {
        List<string> lines = new List<string>();
        lines = GetMapFromTexture(textureMap);
        SetDimensions(lines);
        map = new int[mapWidth, mapHeight];
        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                if (lines[y].Length > x)
                {
                    map[x, y] = (int)char.GetNumericValue(lines[y][x]);
                }

            }
        }
    }
    else
    {
        map = new int[mapWidth, mapHeight];
        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                if (x == 0 || y == 0 || x == mapWidth - 1 || y == mapHeight - 1)
                {
                    map[x, y] = 1;
                }
                else
                {
                    map[x, y] = 0;
                }
            }
        }
    }
    return map;
}
```

Figure[38]: Function to design the layout of the map in MapData class.

```
public List<string> GetMapFromTexture(Texture2D texture)
{
    List<string> lines = new List<string>();

    if (textureMap != null)
    {
        for (int y = 0; y < texture.height; y++)
        {
            string newLine = "";

            for (int x = 0; x < texture.width; x++)
            {
                if (texture.GetPixel(x, y) == Color.black)
                {
                    newLine += '1';
                }
                else if (texture.GetPixel(x, y) == Color.white)
                {
                    newLine += '0';
                }
                else if (texture.GetPixel(x, y) == Color.blue)
                {
                    newLine += '9';
                }
                else if (texture.GetPixel(x, y) == Color.green)
                {
                    newLine += '3';
                }
                else
                {
                    newLine += '0';
                }
            }
            lines.Add(newLine);
        }
    }
    return lines;
}
```

Figure[39]: Function to read from a texture.

```
public void CreateGrid(int[,] mapdata)
{
    gridWidth = mapdata.GetLength(0); //first array so x
    gridHeight = mapdata.GetLength(1); //first array so y;

    gridNodes = new Node[gridWidth, gridHeight];

    for (int y = 0; y < gridHeight; y++)
    {
        for (int x = 0; x < gridWidth; x++)
        {
            //change nodetype depending on what number we gave
            NodeType type = (NodeType)mapdata[x, y];
            //Node world position
            Vector3 worldPosition = new Vector3(x, 0, y);
            //create a new node
            Node newNode = new Node(x, y, type, worldPosition)
            //place it in the grid of nodes
            gridNodes[x, y] = newNode;
        }
    }

    for (int y = 0; y < gridHeight; y++)
    {
        for (int x = 0; x < gridWidth; x++)
        {
            //pre-calc neighbours.
            gridNodes[x, y].neighbours = GetNeighbours(x, y);
        }
    }
}
```

Figure[40]: Function that instantiates our grid of nodes in GridManager class.

```
public void CreateGridVisualisation(GridManager grid)
{
    if (grid == null)
    {
        Debug.LogWarning("GridManager has returned as null");
        return;
    }

    nodesVisualisationData = new NodeVisualisation[grid.GetGridWidth, grid.GetGridHeight];

    foreach (Node node in grid.gridNodes)
    {
        GameObject instance = Instantiate(nodeVisualisationPrefab, Vector3.zero, Quaternion.identity);
        NodeVisualisation nodeVisualisation = instance.GetComponent<NodeVisualisation>();

        if (nodeVisualisation != null)
        {
            nodeVisualisation.CreateNodeVisualisation(node);
            nodesVisualisationData[node.xIndexPosition, node.yIndexPosition] = nodeVisualisation;
            if (nodeVisualisation.gridNode.nodeType == NodeType.Blocked)
            {
                nodeVisualisation.ColorNode(wallColor);
            }
            else if (nodeVisualisation.gridNode.nodeType == NodeType.Open)
            {
                nodeVisualisation.ColorNode(baseColor);
            }
            else if (nodeVisualisation.gridNode.nodeType == NodeType.GoalNode)
            {
                nodeVisualisation.ColorNode(goalColor);
            }
            else if (nodeVisualisation.gridNode.nodeType == NodeType.Grass)
            {
                nodeVisualisation.ColorNode(grassColor);
            }
            else if (nodeVisualisation.gridNode.nodeType == NodeType.Water)
            {
                nodeVisualisation.ColorNode(waterColor);
            }
        }
    }
}
```

Figure[41]: Function that instantiates the node visuals in our scene GridVisualisation class.

C.2 - Visual Aid

```
void ColorNode(Color color, GameObject go)
{
    if (go != null)
    {
        Renderer goRenderer = go.GetComponent<Renderer>();

        if (goRenderer != null)
        {
            goRenderer.material.color = color;
        }
    }
}
```

```
public void ColorNode(Color color)
{
    ColorNode(color, tile);
}
```

Figure [42]: ColorNode function from NodeVisualisation class.

```
public void ColorNodes(List<Node> nodelist, Color color)
{
    foreach (Node node in nodelist)
    {
        if (node != null)
        {
            NodeVisualisation nodeVisualisation = nodesVisualisationData[node.xIndexPosition, node.yIndexPosition];

            if (nodeVisualisation != null)
            {
                nodeVisualisation.ColorNode(color);
            }
        }
    }
}
```

Figure [43]: ColorNodes Function from GridVisualisation Class.

```
public void EnableObject(GameObject go, bool state)
{
    go.SetActive(state);
}

2 references
public void ArrowPosition()
{
    if (gridNode.nodeType != NodeType.Blocked)
    {
        Vector3 directionToParent = (gridNode.nodeParent.nodeWorldPosition - gridNode.nodeWorldPosition).normalized;
        arrow.transform.rotation = Quaternion.LookRotation(directionToParent);
    }
}
```

Figure[44]: ArrowPosition function from NodeVisualisation class.

```
public void ChangePositionOfArrow()
{
    foreach(var nodeVisual in nodesVisualisationData)
    {
        nodeVisual.ArrowPosition();
    }
}
```

Figure[45]: ChangePositionOfArrow function in GridVisualisation class.

```
public void ResetGridVisualisation()
{
    foreach (NodeVisualisation nodeVisualisation in nodesVisualisationData)

        if (nodeVisualisation.gridNode.nodeType == NodeType.Blocked)
        {
            nodeVisualisation.ColorNode(wallColor);
        }
        else if (nodeVisualisation.gridNode.nodeType == NodeType.Open)
        {
            nodeVisualisation.ColorNode(baseColor);
        }
        else if (nodeVisualisation.gridNode.nodeType == NodeType.GoalNode)
        {
            nodeVisualisation.ColorNode(goalColor);
        }
        else if (nodeVisualisation.gridNode.nodeType == NodeType.Grass)
        {
            nodeVisualisation.ColorNode(grassColor);
        }
        else if (nodeVisualisation.gridNode.nodeType == NodeType.Water)
        {
            nodeVisualisation.ColorNode(waterColor);
        }
}
```

Figure [46]: Reset the grid colours, function in GridVisualisation class.

C.3 - Dynamic Map/Pathfinding

```
if (Input.GetMouseButtonDown(0))
{
    if (Physics.Raycast(ray, out hit))
    {
        Debug.Log(hit.collider.gameObject.tag);
        if (hit.collider.gameObject.tag == "Node")
        {
            NodeVisualisation nodeVisualisation = hit.collider.gameObject.GetComponentInParent<NodeVisualisation>();
```

Figure [47]: Raycast code implementation.

```
void LeftMouseClickToAlterNodeTerrain()
{
    if (Input.GetMouseButtonDown(0))
    {
        if (Physics.Raycast(ray, out hit))
        {
            Debug.Log(hit.collider.gameObject.tag);
            if (hit.collider.gameObject.tag == "Node")
            {
                NodeVisualisation nodeVisualisation = hit.collider.gameObject.GetComponentInParent<NodeVisualisation>();
                if (nodeVisualisation != goalNode)
                {

                        switch (terrainIndex)
                        {
                            case 0:
                                ChangeTileTerrain(nodeVisualisation, NodeType.Open, false);
                                RecalculateUnitPath();
                                break;
                            case 1:
                                ChangeTileTerrain(nodeVisualisation, NodeType.Blocked, true);
                                SearchUnitPathRecalculate(nodeVisualisation);
                                break;
                            case 2:
                                ChangeTileTerrain(nodeVisualisation, NodeType.Grass, false);
                                RecalculateUnitPath();
                                break;
                            case 3:
                                ChangeTileTerrain(nodeVisualisation, NodeType.Water, false);
                                RecalculateUnitPath();
                                break;
                            default:
                                break;
                        }
                }
            }
        }
    }
}
```

Figure [48]: Alter terrain on click function in scene and minheap controller class.

```
void ChangeTileTerrain(NodeVisualisation node, NodeType nodeType, bool wallStatus)
{
    node.gridNode.nodeType = nodeType;
    node.EnableObject(node.wall, wallStatus);
    gridVisualisation.ResetGridVisualisation();
}
```

Figure [49]: ChangeTileTerrain function in scene and minheap controller class

```
void RecalculateUnitPath()
{
    ClearTextFromDisplay();
    for (int i = 0; i < unitData.Count; i++)
    {
        unitData[i].UnitFindPath(goalNode.transform, algorithmIndex, heuristicIndex);

        AddMessageToTextBox(unitData[i].ReturnUnitMessage(), i);
        if (PathfindingVisualisationAid)
        {
            if (!(unitData.Count > 1))
            {
                unitData[i].UnitSearchVisualisation();
            }
            unitData[i].UnitPathVisualisation();
            gridVisualisation.ChangeToGoalNodeColourOnly(goalNode);
        }
    }
    UpdateScrollBarToBottom();
}

void SearchUnitPathRecalculate(NodeVisualisation node)
{
    ClearTextFromDisplay();
    for (int i = 0; i < unitData.Count; i++)
    {
        if (unitData[i].SearchPathChangedNode(node.gridNode))
        {
            unitData[i].UnitFindPath(goalNode.transform, algorithmIndex, heuristicIndex);
        }
        AddMessageToTextBox(unitData[i].ReturnUnitMessage(), i);
        if (PathfindingVisualisationAid)
        {
            if (!(unitData.Count > 1))
            {
                unitData[i].UnitSearchVisualisation();
            }
            unitData[i].UnitPathVisualisation();
            gridVisualisation.ChangeToGoalNodeColourOnly(goalNode);
        }

    }
    UpdateScrollBarToBottom();
}
```

Figure [50]: Recaulate unit path functions in scene (basic pathfinding) and minheap controller.

```
switch (terrainIndex)
{
    case 0:
    if (nodeVisualisation.gridNode.nodeType != NodeType.Open)
    {
        ChangeTileTerrain(nodeVisualisation, NodeType.Open, false);
        UpdateMap();
    }
        break;
    case 1:
    if (nodeVisualisation.gridNode.nodeType != NodeType.Blocked)
    {
        ChangeTileTerrain(nodeVisualisation, NodeType.Blocked, true);
        UpdateMap();
    }
        break;
    case 2:
    if (nodeVisualisation.gridNode.nodeType != NodeType.Grass)
    {
        ChangeTileTerrain(nodeVisualisation, NodeType.Grass, false);
        UpdateMap();
    }
    break;
    case 3:
    if (nodeVisualisation.gridNode.nodeType != NodeType.Water)
    {
        ChangeTileTerrain(nodeVisualisation, NodeType.Water, false);
        UpdateMap();
    }
    break;
}
```

Figure [51]: Flowfield version of the switch for changing terrain in Flowfield controller.

```
void ChangeTileTerrain(NodeVisualisation node, NodeType nodeType, bool wallStatus)
{
    node.gridNode.nodeType = nodeType;
    node.EnableObject(node.wall, wallStatus);
    node.EnableObject(node.arrow, PathfindingVisualisationAid);
    gridVisualisation.ResetGridVisualisation();
}
```

Figure [52]: Change tile terrain for Flowfield controller.

```
void UpdateMap()
{
    flowfieldPathfinding.FlowfieldPath(goalNode.transform.position, PathfindingVisualisationAid, heuristicIndex);
    ResetUnitHasReachedTarget();
    foreach(var unit in unitData)
    {
        Node currentNode = grid.GetNodeFromWorldPoint(unit.transform.position);
        currentNode.UnitAbove = true;
    }
}
```

Figure [53]: UpdateMap function in Flowfield controller.

```
void RightMouseClickToChangeGoalNodePositon()
{
    //Right click
    if (Input.GetMouseButtonDown(1))
    {
        if (Physics.Raycast(ray, out hit))
        {
            Debug.Log(hit.collider.gameObject.tag);
            //added mesh chollider to tile to make sure we are able to hit the tile plane.
            if (hit.collider.gameObject.tag == "Node")
            {
                NodeVisualisation nodeVisualisation = hit.collider.gameObject.GetComponentInParent<NodeVisualisation>();
                if (nodeVisualisation.gridNode.nodeType != NodeType.Blocked)
                {
                    ChangeTileToGoalNode(nodeVisualisation);
```

Figure [54]: Right click to apply new goal to map in all controller classes.

```
void MiddleMouseClickToSpawnUnit()
{
    if (Input.GetMouseButtonDown(2))
    {
        if (Physics.Raycast(ray, out hit))
        {
            Debug.Log(hit.collider.gameObject.tag);
            //added mesh chollider to tile to make sure we are able to hit the tile plane.
            if (hit.collider.gameObject.tag == "Node")
            {
                NodeVisualisation nodeVisualisation = hit.collider.gameObject.GetComponentInParent<NodeVisualisation>();
                if (nodeVisualisation.gridNode.nodeType != NodeType.Blocked && nodeVisualisation.gridNode.nodeType != NodeType.GoalNode)
                {
                    InstantiateUnit(nodeVisualisation);
                }
            }
        }
    }
}
```

Figure [55]: Middle button click function in all scene controllers.

```
void InstantiateUnit(NodeVisualisation node)
{
    Unit unit = Instantiate(unitPrefab, node.transform.position + unitPrefab.transform.position, Quaternion.identity);
    unit.SetUnitPositionInGrid(node.gridNode.xIndexPosition, node.gridNode.yIndexPosition);
    unit.InitiatePathfinding(grid, gridVisualisation);
    unitData.Add(unit);
    unit.UnitFindPath(goalNode.transform, algorithmIndex, heuristicIndex);
    AddMessageToTextBox(unit.ReturnUnitMessage(), unitData.Count - 1);
    UpdateScrollBarToBottom();
    if (PathfindingVisualisationAid)
    {
        if(!(unitData.Count > 1))
        {
            unit.UnitSearchVisualisation();
        }
        unit.UnitPathVisualisation();
        gridVisualisation.ChangeToGoalNodeColourOnly(goalNode);
    }
}
```

Figure [56]: Unit instantiation function scene (basic pathfinding) and min heap controllers.

```
void InstantiateUnit(NodeVisualisation node)
{
    FlowfieldUnit unit = Instantiate(unitPrefab, node.transform.position + unitPrefab.transform.position, Quaternion.identity);
    unit.InstaniateUnit(node.gridNode.xIndexPosition, node.gridNode.yIndexPosition, grid, unitData.Count);
    unitData.Add(unit);
}
```

Figure [57]: Unit instantiation function in Flowfield controller.

C.4 - User Interface

```
public void SetCurrentAlgorithmFromDropdown(int algorithmindex)
{
    algorithmIndex = algorithmindex;
}
0 references
public void SetTerrainTypeFromDropdown(int terrainindex)
{
    terrainIndex = terrainindex;
}

0 references
public void SetHeuristicAlgorithmFromDropdown(int heuristicindex)
{
    heuristicIndex = heuristicindex;
}
```

Figure [58]: Apply the index from the selected method from the UI dropdown.

```
public void ChangeCustomDistanceOnClick()
{
    if (distanceStraight.text != "" && distanceDiagnol.text != "")
    {
        grid.SetCustomDistanceValues(float.Parse(distanceDiagnol.text), float.Parse(distanceStraight.text));
    }
}
```

Figure [59]: Change the custom heuristic values on button click.

```
public void ChangeUnitPositionOnButtonClick()
{

    if (xUnitInput.text != "" && yUnitInput.text != "")
    {
        xUnitInputValue = int.Parse(xUnitInput.text);
        yUnitInputValue = int.Parse(yUnitInput.text);

        if (grid.IsWithinBounds(xUnitInputValue, yUnitInputValue))
        {
            NodeVisualisation node = gridVisualisation.nodesVisualisationData[xUnitInputValue, yUnitInputValue];
            if (node.gridNode.nodeType != NodeType.Blocked)
            {
                unitData[0].ChangeUnitPositionWithoutUsingSpawnPosition(node.gridNode.xIndexPosition, node.gridNode.yIndexPosition);
                unitData[0].ClearList();
                gridVisualisation.ResetGridVisualisation();
                gridVisualisation.ChangeToGoalNodeColourOnly(goalNode);
            }
            else
            {
                Debug.Log("Can't place unit on a blocked node please input a valid walkable node");
            }
        }
        else
        {
            Debug.Log("Number Inputted is out of bounds. Please put a number that corresponds to the size of the map");
        }
    }
    else
    {
        Debug.Log("A value must be given for the unit");
    }
}
```

Figure [60]: Change a unit position when button is clicked.

Flowfield controller has a similar implementation except instead of calling recalculate unit path function it calls upon map update function instead.

```
public void ChangeGoalPositionOnButtonClick()
{
    //If the text box is not blank then crack on otherwise throw debug message.
    if (xGoalInput.text != "" && yGoalInput.text != "")
    {
        xGoalInputValue = int.Parse(xGoalInput.text);
        yGoalInputValue = int.Parse(yGoalInput.text);

        if (grid.IsWithinBounds(xGoalInputValue, yGoalInputValue))
        {
            NodeVisualisation node = gridVisualisation.nodesVisualisationData[xGoalInputValue, yGoalInputValue];
            if (node.gridNode.nodeType != NodeType.Blocked)
            {
                ChangeTileToGoalNode(node);
                RecalculateUnitPath();
            }
            else
            {
                Debug.Log("Can't place goal on a blocked node please input a valid walkable node");
            }
        }
        else
        {
            Debug.Log("Number Inputted is out of bounds. Please put a number that corresponds to the size of the map");
        }
    }
    else
    {
        Debug.Log("A value must be given");
    }
}
```

Figure [61]: Change goal position on button click to the co-ordinates provided.

```
public void ToggleVisualisationAid(bool toggle)
{
    PathfindingVisualisationAid = toggle;

    if (PathfindingVisualisationAid)
    {
        //shouldn't need to do this but just incase
        gridVisualisation.ResetGridVisualisation();
        foreach (var unit in unitData)
        {
            if(!(unitData.Count > 1))
            {
                unit.UnitSearchVisualisation();
            }
            unit.UnitPathVisualisation();
            gridVisualisation.ChangeToGoalNodeColourOnly(goalNode);
        }
    }
    else
    {
        gridVisualisation.ResetGridVisualisation();
        gridVisualisation.ChangeToGoalNodeColourOnly(goalNode);
    }
}
```

Figure [62]: Pathfinding and minheap implementation when visual aid toggle is clicked.

```
public void VisualAidShowNodeArrows(bool visualaid)
{
    PathfindingVisualisationAid = visualaid;
    gridVisualisation.ShowNodeArrows(visualaid);
}
```

Figure [63]: Flowfield implementation when visual aid toggle is clicked.

```
3 references
void AddMessageToTextBox(string unitmessage, int unitnumber) {
    unitInfoDisplay.text += "Unit " + unitnumber + " - " + unitmessage;
    unitInfoDisplay.text += "\n";

}
2 references
void ClearTextFromDisplay()
{
    if (unitData.Count > 1)
    {
        unitInfoDisplay.text = "";
    }
}
3 references
void UpdateScrollBarToBottom()
{
    Canvas.ForceUpdateCanvases();
    scrollObject.verticalNormalizedPosition = 0.0f;
}
```

Figure [64]: Functionality for changing the text, clearing displaying and updating scroll wheel.

C.5 - Unit & Unit Movement

```
public void UnitFindPath(Transform goalposition, int algorithmindex, int heuristicindex)
{
    goalWorldPosition = goalposition;
    algorithmIndex = algorithmindex;
    heuristicIndex = heuristicindex;

    if (goalposition != null)
    {
        StopCoroutine("MoveUnitAcrossPath");
        path = pathFinding.FindPath(transform.position, goalWorldPosition.position, algorithmIndex, heuristicIndex, out unitMessagePathfinding);
        if (path != null)
        {
            //StartCoroutine(MoveUnitAcrossPath(algorithmindex, heuristicindex));
            StartCoroutine("MoveUnitAcrossPath");
        }
    }
}
```

Figure [65]: UnitFindPath function in minheap and unit class.

```
IEnumerator MoveUnitAcrossPath()
{
    Vector3 currentNodeWorldPosition = path[0].nodeWorldPosition;
    IndexInPath = 0;

    while (true)
    {
        if (transform.position == currentNodeWorldPosition)
        {
            IndexInPath++;
            if (IndexInPath > path.Count - 1)
            {
                yield break;
            }
            currentNodeWorldPosition = path[IndexInPath].nodeWorldPosition;
        }

        if (path[IndexInPath].nodeType == NodeType.Blocked)
        {
            path = pathFinding.FindPath(transform.position, goalWorldPosition.position, algorithmIndex, heuristicIndex, out unitMessagePathfinding);
            IndexInPath = 0;
            yield return null;
        }
        currentNodeWorldPosition.y = transform.position.y;
        transform.position = Vector3.MoveTowards(transform.position, currentNodeWorldPosition, unitSpeed);
        yield return null;
    }
}
```

Figure [66]: MoveUnitAcrossPath function in minheap and unit class.

```
public void UnitMovementStart(Node node)
{
    StartCoroutine("MoveUnitToNode", node);
}
```

```
IEnumerator MoveUnitToNode(Node currentnode)
{
    bool UnitPastCurrentNode = false;
    currentnode.UnitAbove = true;
    SetReachedTarget(false);
    currentnode.nodeParent.UnitAbove = true;
    Vector3 tempNodePos = new Vector3(currentnode.nodeParent.nodeWorldPosition.x,
        transform.position.y, currentnode.nodeParent.nodeWorldPosition.z);
    int count = 0;

    while (true)
    {
        if(count > 30)
        {
            SetReachedTarget(true);
            yield break;
        }
        transform.position = Vector3.MoveTowards(transform.position ,tempNodePos, unitSpeed);
        if(!UnitPastCurrentNode)
        {
            if(gridFlowfield.CheckIfUnitOnNode(transform.position, currentnode))
            {
                gridFlowfield.RecalculateNeighbours(currentnode);
                UnitPastCurrentNode = true;
            }
        }
        if (transform.position == tempNodePos)
        {
            SetReachedTarget(true);
            yield break;
        }
        count++;
        yield return null;
    }
}
```

Figure [67]: Unit movement for Flowfield unit class.

C.6 - Minimum Binary Heap

```
public interface IHeapItem<T> : IComparable<T>
{
    21 references
    int HeapIndex
    {
        get;
        set;
    }
}

public class MinHeap<T> where T : IHeapItem<T>

    public class Node : IHeapItem<Node>
```

Figure [68]: IHeapItem interface has specific methods that must have an implementation if inheriting the IHeapItem. MinHeap class stating any data type must inherit from IHeapItem and Node class inheriting the IHeapItem.

```
public int HeapIndex
{
    get { return heapIndex;  }
    set { heapIndex = value; }
}
```

Figure [69]: HeapIndex functionality provided in the Node class.

```
public int CompareTo(Node nodeToCompare)
{
    int compare = 0;
    if (!float.IsInfinity(fCost))
    {
        compare = fCost.CompareTo(nodeToCompare.fCost);
        if (compare == 0)
        {
            compare = hCost.CompareTo(nodeToCompare.hCost);
        }
    }
    else if (!float.IsInfinity(hCost))
    {
        compare = hCost.CompareTo(nodeToCompare.hCost);
    }
    else if (!float.IsInfinity(gCost))
    {
        compare = gCost.CompareTo(nodeToCompare.gCost);
    }
    return -compare;
}
```

Figure [70]: CompareTo functionality provided in the Node Class.

```
public void Add(T item)
{
    //Throw exception if there is a error.
    if (currentItemAmount == data.Length)
    {
        throw new IndexOutOfRangeException();
    }

    /* set the items current position in the array*/
    item.HeapIndex = currentItemAmount;
    /*store the item in the array using the current item count*/
    data[currentItemAmount] = item;
    /*increment the item count*/
    currentItemAmount++;

    /*Recalculate the heap to see which position the new item
    should be in in the heap.*/
    ReCalculateUp(item);
}
```

Figure [71]: Function that adds a node to the minheap in the min heap class.

```
private void ReCalculateUp(T item)
{
    /*We want to swap when the item isn't the top of the binary tree and
    and the new item added is less than it's parent.*/
    while (!IsRoot(item.HeapIndex) && data[item.HeapIndex].CompareTo(GetParent(item.HeapIndex)) > 0)
    {
        //Swaps the positions of the items;
        Swap(data[item.HeapIndex], GetParent(item.HeapIndex));
    }
}
```

Figure[72]: Recalculate up function in min heap class.

```
private void Swap(T itemA,T itemB)
{
    data[itemA.HeapIndex] = itemB;
    data[itemB.HeapIndex] = itemA;

    int tempItemIndex = itemA.HeapIndex;
    itemA.HeapIndex = itemB.HeapIndex;
    itemB.HeapIndex = tempItemIndex;

}
```

Figure [73]: Swap function in min heap class.

```
public T RemoveFrontItem()
{
    if (currentItemAmount == 0)
    {
        throw new IndexOutOfRangeException();
    }

    T frontItem = data[0];
    data[0] = data[currentItemAmount - 1];
    data[0].HeapIndex = 0;
    currentItemAmount--;
    ReCalculateDown(data[0]);

    return frontItem;
}
```

Figure [74]: Removes the front node from the minheap.

```
void ReCalculateDown(T item)
{
    while (HasLeftChild(item.HeapIndex))
    {
        int changeIndex = GetLeftChildIndex(item.HeapIndex);

        if (HasRightChild(item.HeapIndex)
        && GetLeftChild(item.HeapIndex).CompareTo(GetRightChild(item.HeapIndex)) < 0)
        {
            changeIndex = GetRightChildIndex(item.HeapIndex);
        }
        if (item.CompareTo(data[changeIndex]) >= 0)
        {
            return;
        }

        Swap(item, data[changeIndex]);
    }
}
```

Figure[75]: ReCalculateDown function in minheap class.

C.7 - Heuristic

```
public float GetNodeDistance(Node source, Node target, int heuristicindex)
{
    switch (heuristicindex)
    {
        case 0:
            return PatricksDiagnolDistance(source, target);
        case 1:
            return ManhattenDistance(source, target);
        case 2:
            return EuclideanDistance(source, target);
        case 3:
            return ChebyshevDistance(source, target);
        case 4:
            return OctileDistance(source, target);
        case 5:
            return CustomDiagnolDistance(source, target);
    }
    //This will never be hit but we have to return a value since it will complain
    return Mathf.Infinity;
}
```

Figure [76]: Switch implementation of heuristics.

```
float PatricksDiagnolDistance(Node source, Node target)
{
    int xDistance = Mathf.Abs(source.xIndexPosition - target.xIndexPosition);
    int yDistance = Mathf.Abs(source.yIndexPosition - target.yIndexPosition);

    if (xDistance > yDistance)
    {
        return 1.4f * yDistance + 1.0f * (xDistance - yDistance);
    }
    return 1.4f * xDistance + 1.0f * (yDistance - xDistance);
}
```

Figure [77]: Patricks diagonal heuristic method.

```
float ManhattenDistance(Node source, Node target)
{
    int xDistance = Mathf.Abs(source.xIndexPosition - target.xIndexPosition);
    int yDistance = Mathf.Abs(source.yIndexPosition - target.yIndexPosition);

    return 1.0f * (xDistance + yDistance);
}
```

Figure [78]: Manhattan distance heuristic method.

```
float EuclideanDistance(Node source, Node target)
{
    int xDistance = Mathf.Abs(source.xIndexPosition - target.xIndexPosition);
    int yDistance = Mathf.Abs(source.yIndexPosition - target.yIndexPosition);

    return 1.0f * Mathf.Sqrt((xDistance * xDistance) + (yDistance * yDistance));
}
```

Figure [79]: Euclidean distance heuristic method.

```
float ChebyshevDistance(Node source, Node target)
{
    int xDistance = Mathf.Abs(source.xIndexPosition - target.xIndexPosition);
    int yDistance = Mathf.Abs(source.yIndexPosition - target.yIndexPosition);
    ////D * max(dx, dy) + (D2-D) * min(dx, dy)
    return 1.0f * Mathf.Max(xDistance, yDistance) + (1.0f - 1.0f) * Mathf.Min(xDistance, yDistance);
}
```

Figure [80]: Chebyshev distance heuristic method.

```
float OctileDistance(Node source, Node target)
{
    int xDistance = Mathf.Abs(source.xIndexPosition - target.xIndexPosition);
    int yDistance = Mathf.Abs(source.yIndexPosition - target.yIndexPosition);
    //1.41421356237 square route of 2
    //D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
    return 1.0f * (xDistance + yDistance) + (1.4f - 2 * 1.0f) * Mathf.Min(xDistance, yDistance);
}
```

Figure [81]: Octile distance heuristic method.

```
float CustomDiagnolDistance(Node source, Node target)
{
    int xDistance = Mathf.Abs(source.xIndexPosition - target.xIndexPosition);
    int yDistance = Mathf.Abs(source.yIndexPosition - target.yIndexPosition);

    if (xDistance > yDistance)
    {
        return customDistanceDiagnol * yDistance + customDistanceStraight * (xDistance - yDistance);
    }
    return customDistanceDiagnol * xDistance + customDistanceStraight * (yDistance - xDistance);
}
```

Figure [82]: Custom distance heuristic method.

C.8 - Basic Pathfinding Algorithms

```
public List<Node> FindPath(Vector3 startposition, Vector3 goalposition, int algorithmindex, int heuristicindex, out string unitmessage)
{
    ResetNodeParentgCostAndhCost();
    ClearLists();

    Node startNode = grid.GetNodeFromWorldPoint(startposition);
    Node goalNode = grid.GetNodeFromWorldPoint(goalposition);
    Node currentNode;

    Stopwatch timer = new Stopwatch();

    string pathfindingUsed = "";
    startNode.gCost = 0;
    startNode.hCost = grid.GetNodeDistance(startNode, goalNode, heuristicindex) + (int)startNode.nodeType;
    openList.Add(startNode);
    timer.Start();

    while (openList.Count > 0)
    {
        switch (algorithmindex)
```

Figure [83]: Top half of FindPath function before the switch.

```
void ResetNodeParentgCostAndhCost()
{
    for (int x = 0; x < grid.GetGridWidth; x++)
    {
        for (int y = 0; y < grid.GetGridHeight; y++)
        {
            grid.gridNodes[x, y].Reset();
        }
    }
}
```

Figure [84]: Reset the nodes back to default values.

```
public Node GetNodeFromWorldPoint(Vector3 worldposition)
{
    int x = Mathf.RoundToInt(worldposition.x);
    int y = Mathf.RoundToInt(worldposition.z);

    return gridNodes[x, y];
}
```

Figure [85]: GetNodeFromWorldPoint function in GridManager class.

```
            if (openList.Contains(goalNode))
            {
            pathList = GetPathNodes(goalNode);
            timer.Stop();
            unitmessage = ((pathfindingUsed)
                + (HeuristicUsed(heuristicindex))
                + ("Elapsed time = ")
                + (timer.Elapsed.TotalMilliseconds).ToString()
                + " milliseconds , Nodes Explored = "
                + closedList.Count.ToString()
                + ", Nodes To Goal = "
                + pathList.Count.ToString());


            return pathList;
            }
        }
        unitmessage = ("Path is blocked, no path possible to goal");
        return null;
    }
```

Figure [86]: Bottom of FindPath function after switch.

```
List<Node> GetPathNodes(Node goalnode)
{
    List<Node> path = new List<Node>();
    if (goalnode == null)
    {
        return path;
    }

    path.Add(goalnode);
    Node currentNode = goalnode.nodeParent;
    while(currentNode != null)
    {
        path.Insert(0, currentNode); //saves us haivng to reverse if we use add
        currentNode = currentNode.nodeParent;
    }

    return path;
}
```

Figure [87]: Retrace the path from the goal node and return a list.

```
case 0:
    currentNode = openList[0];
    openList.Remove(currentNode);
    AddCurrentNodeToCloseList(currentNode);
    ExpandBreadthFirstSearchOpenList(currentNode);
    pathfindingUsed = "Breadth First Search with ";
    break;
```

```
void ExpandBreadthFirstSearchOpenList(Node node)
    {
        if (node != null)
        {
            foreach (Node neighbour in node.neighbours)
            {
                if (neighbour.nodeType != NodeType.Blocked
                    && !closedList.Contains(neighbour)
                    && !openList.Contains(neighbour))
                {
                    neighbour.nodeParent = node;
                    openList.Add(neighbour);
                }
            }
        }
    }
```

Figure [88]: Breadth-first-search switch and code implementation.

```
case 1:
    currentNode = openList.Last();
    openList.Remove(currentNode);
    AddCurrentNodeToCloseList(currentNode);
    currentNode.neighbours.Reverse();
    ExpandDepthFirstSearchOpenList(currentNode);
    currentNode.neighbours.Reverse();
    pathfindingUsed = "Depth First Search with ";
    break;

void ExpandDepthFirstSearchOpenList(Node node)
{
    if (node != null)
    {
        foreach (var neighbour in node.neighbours)
        {
            if (neighbour.nodeType != NodeType.Blocked
                && !closedList.Contains(neighbour)
               && !openList.Contains(neighbour))
            {
                neighbour.nodeParent = node;
                openList.Add(neighbour);
            }
        }
    }
}
```

Figure [89]: Depth-first-search switch and code implementation.

```
case 2:
    currentNode = openList[0];

    for (int j = 1; j < openList.Count; j++)
    {
        if (openList[j].gCost < currentNode.gCost)
        {
            currentNode = openList[j];
        }
    }

    openList.Remove(currentNode);
    AddCurrentNodeToCloseList(currentNode);
    ExpandDijkstraOpenList(currentNode, heuristicindex);
    pathfindingUsed = "Dijkstra with ";
    break;
```

```
void ExpandDijkstraOpenList(Node node, int heuristicindex)
{
    if (node != null)
    {
        foreach (Node neighbour in node.neighbours)
        {
            if(neighbour.nodeType == NodeType.Blocked || closedList.Contains(neighbour))
            {
                continue;
            }

            float distanceToNeighbor = node.gCost + grid.GetNodeDistance(node, neighbour, heuristicindex) + (int)neighbour.nodeType;

            if (distanceToNeighbor < neighbour.gCost || !openList.Contains(neighbour))
            {
                neighbour.gCost = distanceToNeighbor;
                neighbour.nodeParent = node;

                if (!openList.Contains(neighbour))
                {
                    openList.Add(neighbour);
                }
            }
        }
    }
}
```

Figure [90]: Dijkstra switch and code implementation.

```
case 3:
    currentNode = openList[0];

    for (int j = 1; j < openList.Count; j++)
    {
        if (openList[j].hCost < currentNode.hCost)
        {
            currentNode = openList[j];
        }
    }

    openList.Remove(currentNode);
    AddCurrentNodeToCloseList(currentNode);
    ExpandGreedyBestFirstSearchOpenList(currentNode, goalNode, heuristicindex);
    pathfindingUsed = "Greedy Best First Search with ";
    break;
```

```
void ExpandGreedyBestFirstSearchOpenList(Node node, Node goalnode, int heuristicindex)
{
    if (node != null)
    {
        foreach (Node neighbour in node.neighbours)
        {
            if (neighbour.nodeType != NodeType.Blocked
                    && !closedList.Contains(neighbour)
                   && !openList.Contains(neighbour))
            {
                neighbour.hCost = grid.GetNodeDistance(neighbour, goalnode, heuristicindex)+(int)neighbour.nodeType;
                neighbour.nodeParent = node;
                openList.Add(neighbour);
            }
        }
    }
}
```

Figure [91]: Greedy-Best-first-search switch and code implementation.

```
case 4:
    currentNode = openList[0];

    for (int j = 1; j < openList.Count; j++)
    {
        if (openList[j].fCost < currentNode.fCost)
        {
            currentNode = openList[j];
        }
    }

    openList.Remove(currentNode);
    AddCurrentNodeToCloseList(currentNode);
    ExpandAStarOpenList(currentNode, goalNode, heuristicindex);
    pathfindingUsed = "A* Pathfinding with ";
    break;
```

```
void ExpandAStarOpenList(Node node, Node goalnode, int heuristicindex)
{
    if (node != null)
    {
        foreach (Node neighbour in node.neighbours)
        {
            if (neighbour.nodeType == NodeType.Blocked || closedList.Contains(neighbour))
            {
                continue;
            }

            float distanceToNeighbor = node.gCost + grid.GetNodeDistance(node, neighbour, heuristicindex) + (int)neighbour.nodeType;

            if (distanceToNeighbor < neighbour.gCost || !openList.Contains(neighbour))
            {
                neighbour.gCost = distanceToNeighbor;
                neighbour.hCost = grid.GetNodeDistance(neighbour, goalnode, heuristicindex) + (int)neighbour.nodeType;
                neighbour.nodeParent = node;

                if (!openList.Contains(neighbour))
                {
                    openList.Add(neighbour);
                }
            }
        }
    }
}
```

Figure [92]: A switch and code implementation.*

```
void AddCurrentNodeToCloseList(Node node)
{
    if (!closedList.Contains(node))
    {
        closedList.Add(node);
    }
}
```

Figure [93]: Function that adds node to closed list.

```
while (openList.Count() > 0)
{
    currentNode = openList.RemoveFrontItem();

    if (!closedList.Contains(currentNode))
    {
        closedList.Add(currentNode);
        nodesExploredCount++;
    }

    switch (algorithmindex)
    {
        case 0:
            ExpandDijkstraOpenList(currentNode, heuristicindex);
            pathfindingUsed = "Dijkstra with ";
            break;
        case 1:
            ExpandBestFirstSearchOpenList(currentNode, goalNode, heuristicindex);
            pathfindingUsed = "Greedy Best First Search with ";
            break;
        case 2:
            ExpandAStarOpenList(currentNode, goalNode, heuristicindex);
            pathfindingUsed = "A* Pathfinding with ";
            break;
        default:
            break;
    }
```

Figure [94]: MinHeap pathfinding class switch implementation.

C.9 - Flowfield Algorithm

```
public void FlowfieldPath(Vector3 goalposition, bool visualaid, int heuristicindex)
{
    ResetNodeParentgCostAndhCost();
    ClearLists();

    Node goalNode = grid.GetNodeFromWorldPoint(goalposition);
    Node currentNode;

    Stopwatch timer = new Stopwatch();

    goalNode.gCost = 0;

    openList.Add(goalNode);
    timer.Start();

    while (openList.Count > 0)
    {

        currentNode = openList[0];
        openList.Remove(currentNode);

        if (!closedList.Contains(currentNode))
        {
            closedList.Add(currentNode);
        }

        CostField(currentNode, heuristicindex);
    }
    grid.IntegrationField();
    if (visualaid)
    {
      gridVisualisation.ChangePositionOfArrow();
    }
}
```

Figure [95]: FlowfieldPath function in Flowfield pathfinding class.

```
void CostField(Node node, int heuristicindex)
{
    if (node != null)
    {
        foreach (Node neighbour in node.neighbours)
        {
            if (neighbour.nodeType == NodeType.Blocked || closedList.Contains(neighbour))
            {
                continue;
            }

            float distanceToNeighbor = node.gCost + grid.GetNodeDistance(node, neighbour, heuristicindex) + (int)neighbour.nodeType;

            if (distanceToNeighbor < neighbour.gCost || !openList.Contains(neighbour))
            {
                neighbour.gCost = distanceToNeighbor;

                if (!openList.Contains(neighbour))
                {
                    openList.Add(neighbour);
                }
            }
        }
    }
}
```

Figure [96]: CostField code implementation.

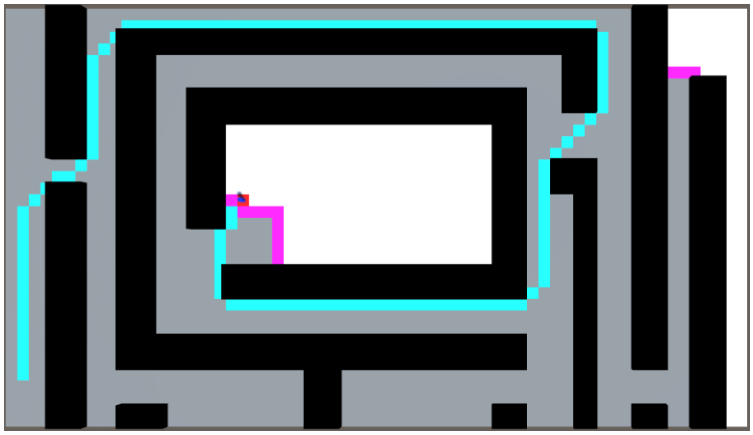
```
public void IntegrationField()
{
    for (int y = 0; y < gridHeight; y++)
    {
        for (int x = 0; x < gridWidth; x++)
        {
            if(gridNodes[x,y].nodeType == NodeType.Blocked)
            {
                continue;
            }

            gridNodes[x, y].nodeParent = gridNodes[x, y];

            foreach (Node neighbour in gridNodes[x, y].neighbours)
            {
                if (neighbour.nodeType == NodeType.Blocked || neighbour.UnitAbove == true)
                {
                    continue;
                }

                if (neighbour.gCost < gridNodes[x, y].nodeParent.gCost)
                {
                    gridNodes[x, y].nodeParent = neighbour;
                }
            }
        }
    }
}
```

Figure [97]: Integration Field code implementation.

Appendix D - Path Behaviour Labyrinth Map Tests

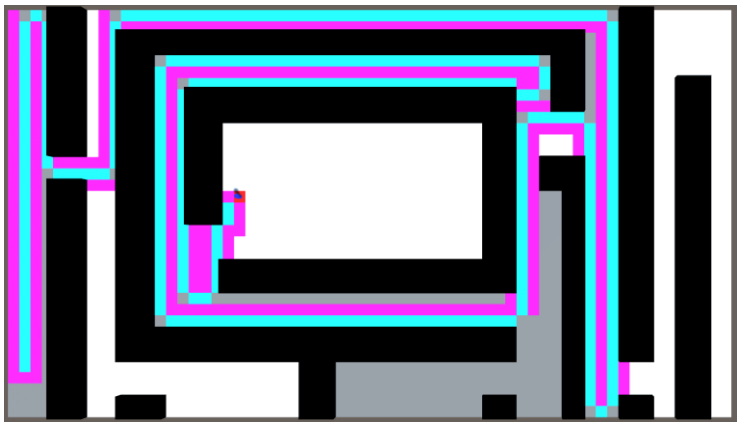
Appendix D shows the most optimal path and the least optimal path for each algorithm from the tests done in **Section 5.4** for the labyrinth map.

D.1 - Breadth-First-search



Figure[98]: Breadth-First-Search path behaviour.

D.2 - Depth-First-Search



Figure[99]: Depth-First-Search path behaviour.

D.3 - Dijkstra

Figure[100]: Dijkstra path behaviour, left is most optimal path right is least optimal path for this algorithm.

D.4 - Greedy-Best-First-Search

Figure[101]: Greedy-Best-First-Search path behaviour, left is most optimal path right is least optimal path for this algorithm.

D.5 - A*

Figure[102]: A path behaviour, left is most optimal path right is least optimal path for this algorithm.*

D.6 - Min-Heap Dijkstra

Figure[103]: Min-Heap Dijkstra path behaviour, left is most optimal path right is least optimal path for this algorithm.

D.7 - Min-Heap Greedy-Best-First-Search

Figure[104]: Min-Heap Greedy-Best-First-Search path behaviour, left is most optimal path right is least optimal path for this algorithm.

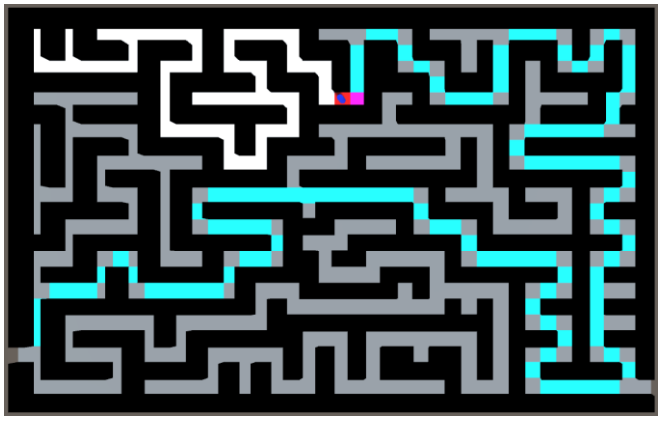
D.8 - Min-Heap A*

Figure[105]: Min-Heap A path behaviour, left is most optimal path right is least optimal path for this algorithm.*

Appendix E - Path Behaviour Maze Map Tests

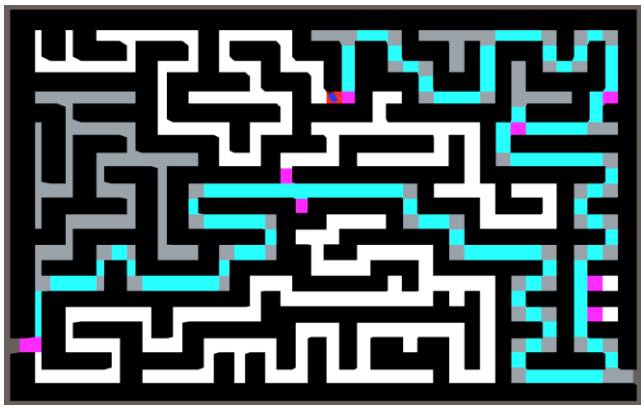
Appendix E shows the most optimal path and the least optimal path for each algorithm from the tests done in **Section 5.5** for the maze map.

E.1 - Breadth-First-search



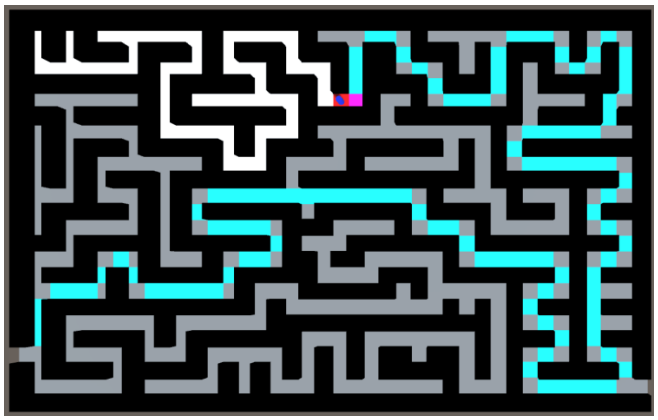
Figure[106]: Breadth-First-Search path behaviour for this algorithm.

E.2 - Depth-First-Search



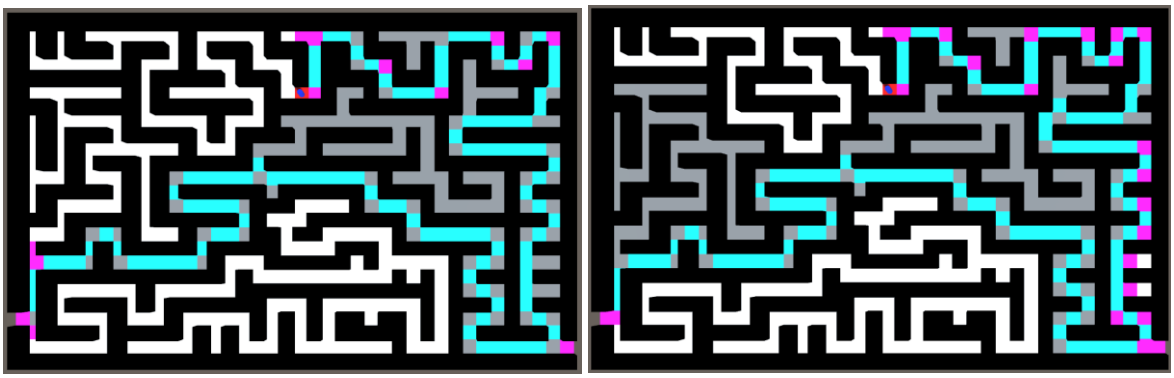
Figure[107]: Depth-First-Search path behaviour for this algorithm.

E.3 - Dijkstra



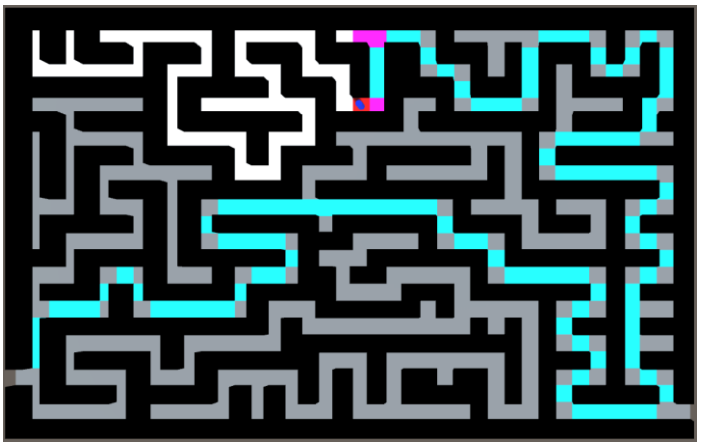
Figure[108]: Dijkstra path behaviour, most optimal path for this algorithm.

E.4 - Greedy-Best-First-Search



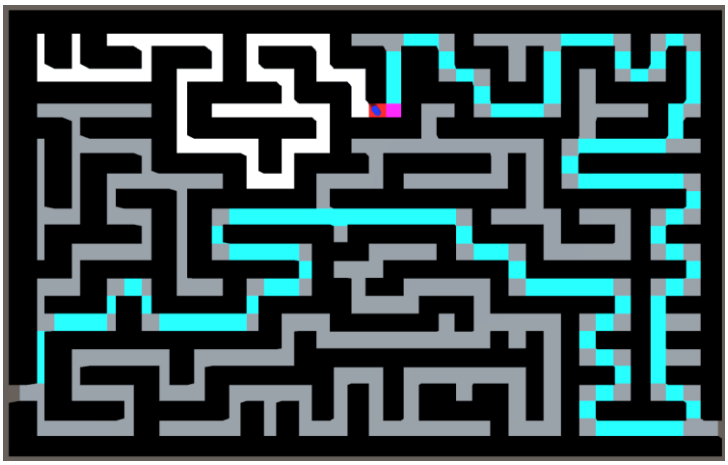
Figure[109]: Greedy-Best-First-Search path behaviour, left is most optimal path right is least optimal path for this algorithm.

E.5 - A*



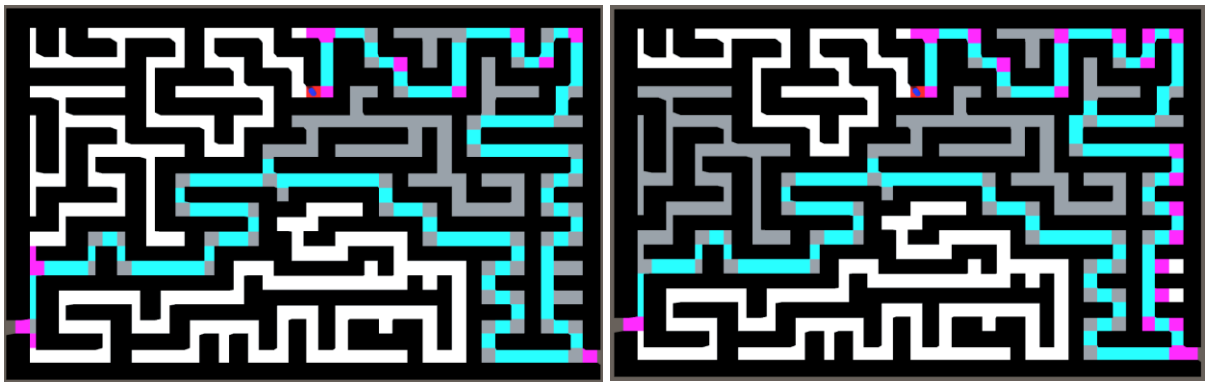
Figure[110]: A path behaviour, most optimal path for this algorithm.*

E.6 - Min-Heap Dijkstra



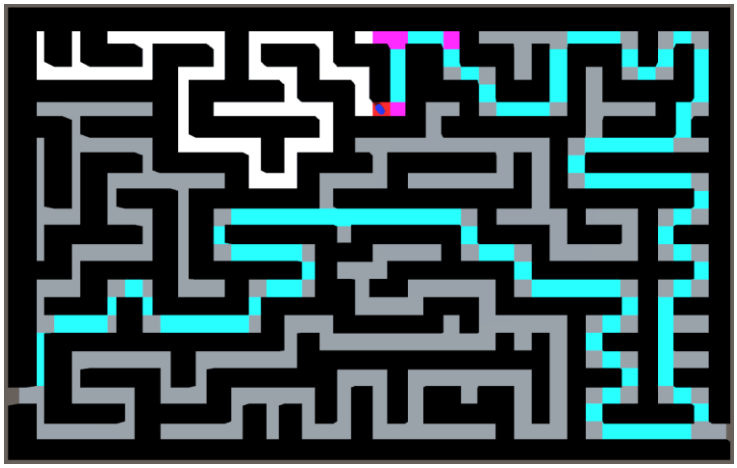
Figure[111]: Min-Heap Dijkstra path behaviour, most optimal path for this algorithm.

E.7 - Min-Heap Greedy-Best-First-Search



Figure[112]: Min-Heap Greedy-Best-First-Search path behaviour, left is most optimal path right is least optimal path for this algorithm.

E.8 - Min-Heap A*

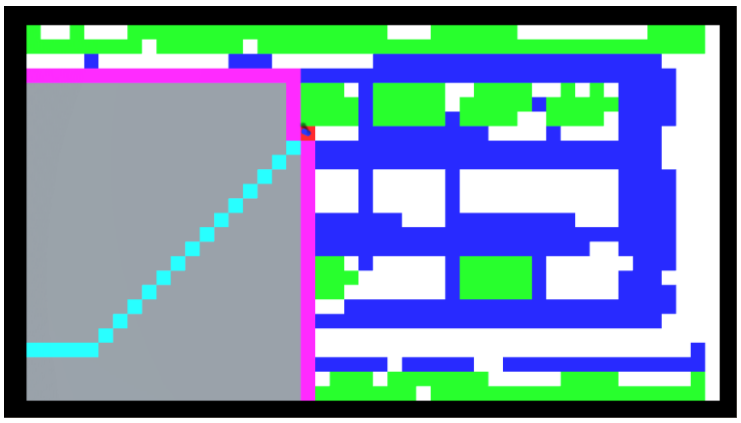


Figure[113]: Min-Heap A path behaviour, most optimal path for this algorithm.*

Appendix F - Path Behaviour Terrain Map Tests

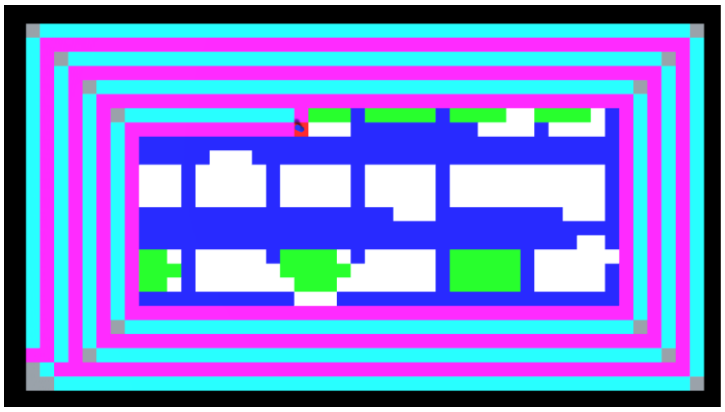
Appendix F shows the most optimal path and the least optimal path for each algorithm from the tests done in **Section 5.6** for the terrain map.

F.1 - Breadth-First-search



Figure[114]: Breadth-First-Search path behaviour.

F.2 - Depth-First-Search

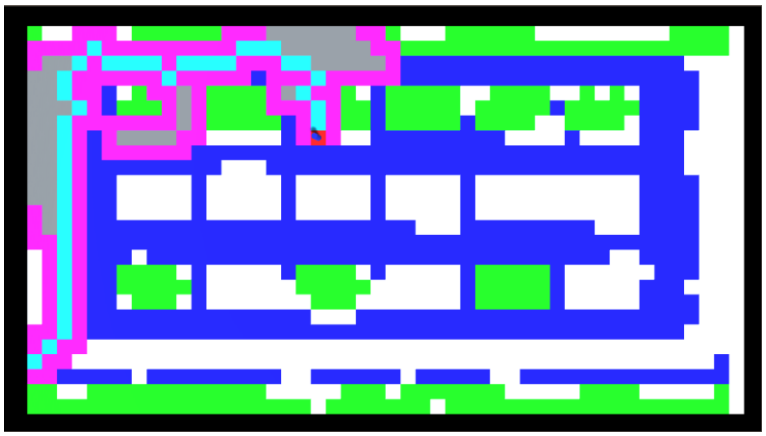


Figure[115]: Depth-First-Search path behaviour.

F.3 - Dijkstra

Figure[116]: Dijkstra path behaviour, most optimal path for this algorithm.

F.4 - Greedy-Best-First-Search



Figure[117]: Greedy-Best-First-Search path behaviour, most optimal path for this algorithm.

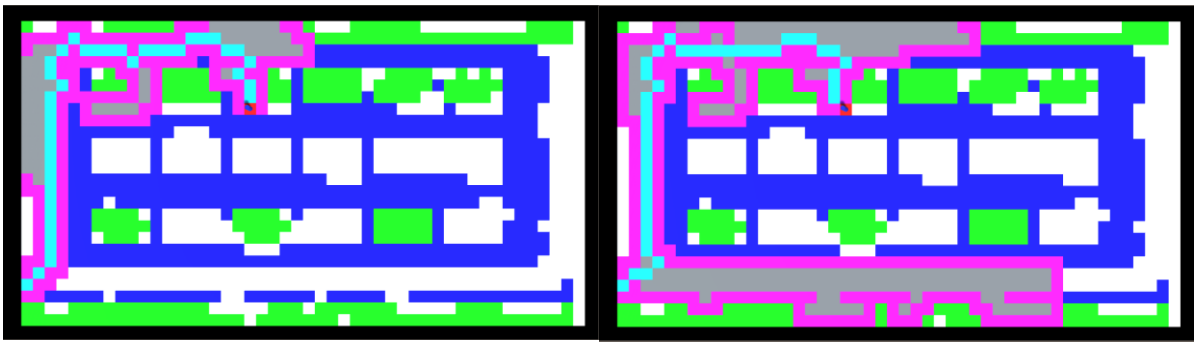
F.5 - A*

Figure[118]: A path behaviour, most optimal path for this algorithm.*

F.6 - Min-Heap Dijkstra

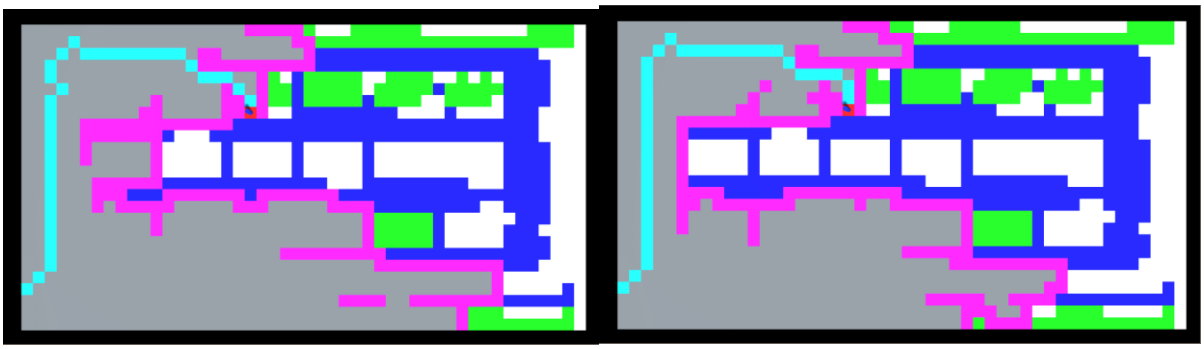
Figure[119]: Min-Heap Dijkstra path behaviour, most optimal path for this algorithm.

F.7 - Min-Heap Greedy-Best-First-Search



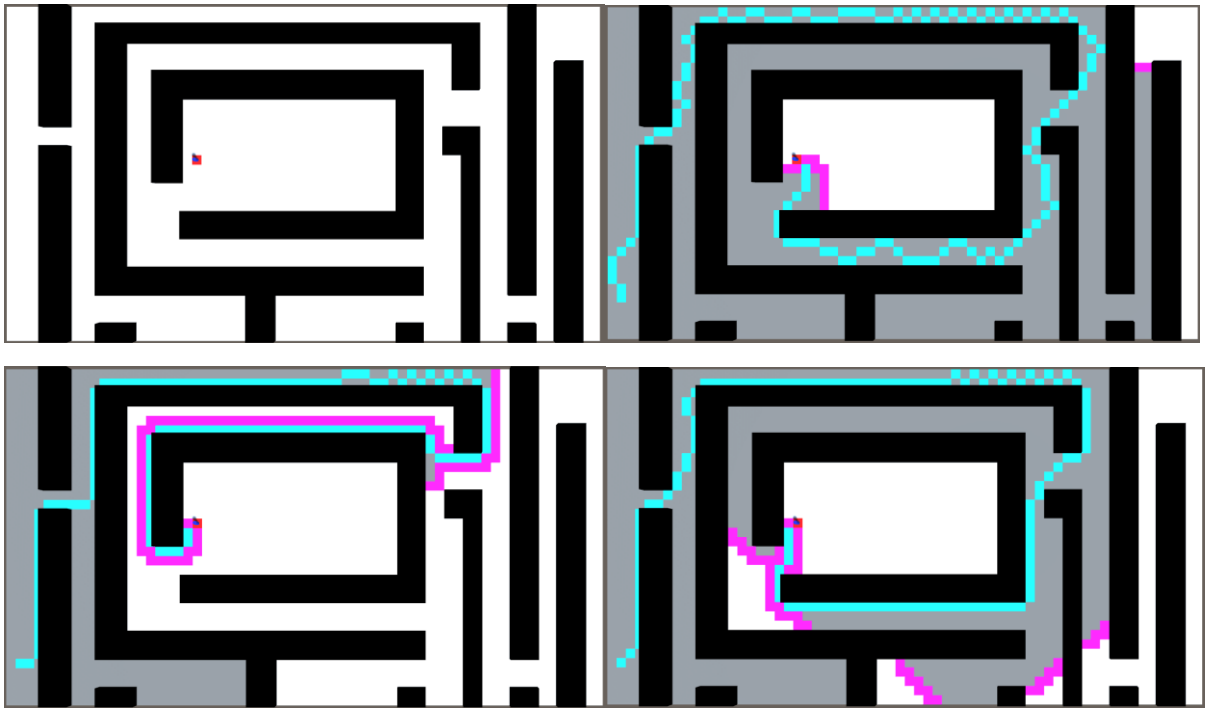
Figure[120]: Min-Heap Greedy-Best-First-Search path behaviour, left is most optimal path right is least optimal path for this algorithm.

F.8 - Min-Heap A*

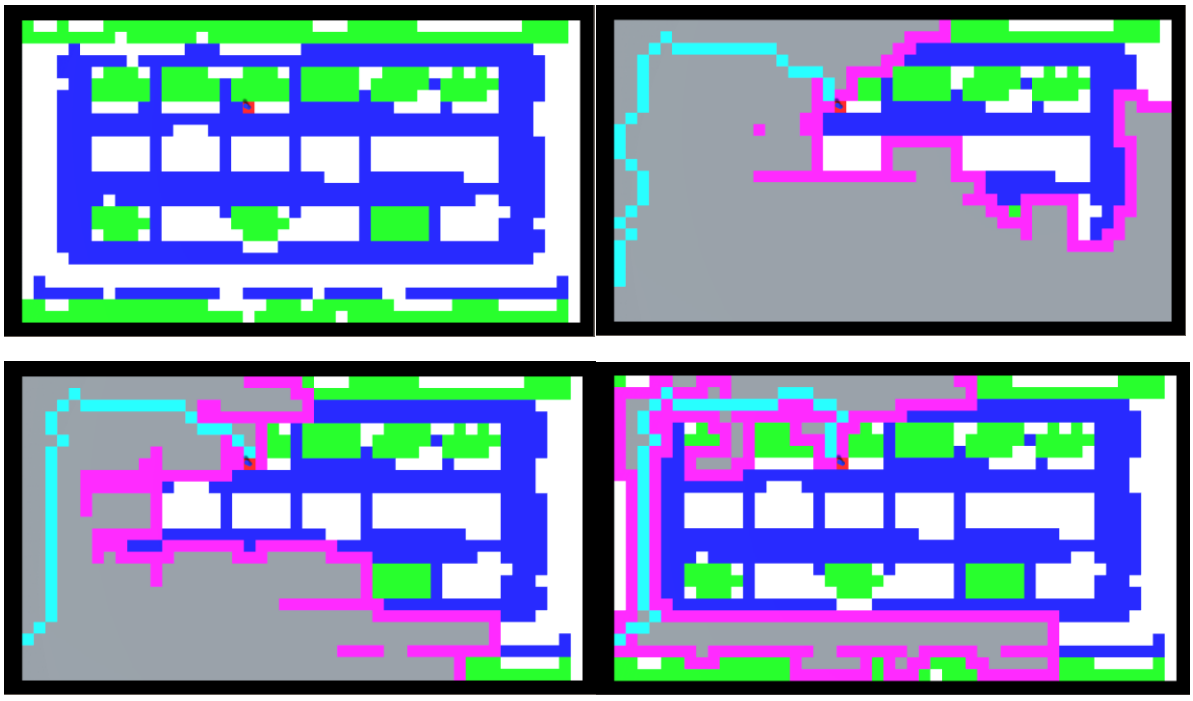


Figure[121]: Min-Heap A path behaviour, left is most optimal path right is least optimal path for this algorithm.*

Appendix G - Min-Heap & Chebyshev Heuristic Problem.



Figure[122]: Labyrinth map, top right Dijkstra, bottom left greedy-best-first-search, bottom right A.*



Figure[123]: Terrain map, top right Dijkstra, bottom left greedy-best-first-search, bottom right A.*

Appendix H - Project Specification for Project (Technical Computing) 2019/20

Student:	**Benjamin Moore – 25024749**
Date:	**24/10/2019**
Supervisor:	**Joe Saunders**
Degree Course:	**Computer Science for Games**
Title of Project:	**Game-based AI Pathfinding Implementation and Optimization**

Elaboration

Many genres of games use AI pathfinding. One of the most common genres you will see this implemented in is real time strategy games, where you would have units that you would want to get from point A to point B in the shortest time possible. But how does this work and how is it done?

This project is to look at AI pathfinding more in depth, investigate the various techniques and understand how it works. It will include the implementation of both basic and advanced AI pathfinding algorithms, optimising the algorithm to improve both the efficiency and performance of the AI pathfinding. The prototype will allow multiple units to traverse through obstacles, finding the shortest path to the end destination. It will have real-time collision avoidance, when its path is blocked in real-time it will re-calculate a new path.

By completing this project, I will:

- Increase my knowledge of games related AI pathfinding and how it is applied to games.
- Understand how to improve the efficiency and performance of the AI pathfinding.
- Apply the knowledge from this project into other fields such as robotics.

Project Aims

The project aims to achieve the following:

- Research Game engines & Learn how to use chosen game engine.
- Research and identify multiple AI pathfinding algorithms.
- Research crowd AI pathfinding & optimization techniques.
- Create a basic grid based 2D map in chosen game engine
- Implement basic AI pathfinding
- Implement dynamic map, dynamic pathfinding and unit to traverse map.
- Implement crowd AI pathfinding
- Improve efficiency and performance of the AI pathfinding.

Project deliverable(s)

This project will deliver:

- Grid based 2d map with a basic user interface that will be able to demonstrate the workings of the basic AI and the crowd AI pathfinding.
- Basic AI pathfinding.
- Dynamic map, dynamic pathfinding and unit traversal
- Crowd AI pathfinding
- Optimisation to the AI pathfinding to make it more efficient.

Task Deadlines

Task	Deadline Date
Game engines analysis & go through chosen game engine tutorials.	Friday 8th of November 2019
Set up source control and project planning tool.	Monday 11th of November 2019
Research AI pathfinding, crowd AI pathfinding & optimization techniques.	Friday 22nd of November 2019
Create basic grid-based 2D Map	Monday 9th of December 2019
Implement basic AI pathfinding	Friday 20th of December 2019
Dynamic map, dynamic pathfinding and unit	Wednesday 1st of January 2020
Implement crowd AI pathfinding	Monday 3rd of February 2020
Improve efficiency & performance of AI pathfinding	Friday 14th of February 2020

Milestone Deadlines

Task	Deadline Date
Information review: • Engine Analysis • AI-path-finding • Crowd AI pathfinding • Optimization Techniques	Thursday 5th of December 2019
Learn how to use unreal/unity, create a grid-based 3d map and basic user interface • Development report	Friday 20th of December 2019
Implement basic AI pathfinding • Development report	Friday 20th of December 2019
Implement Dynamic map, dynamic pathfinding and unit • Development report	January 1st of December 2020
Implement crowd AI pathfinding • Development report	Friday 21st of February 2020
Make the AI pathfinding more efficient • Development report	Friday 21st of February 2020

Module Deadlines

Task	Deadline Date
Find a project supervisor	Friday 11th of October 2019
Project specification and ethics form	3 PM, Friday 25th of October 2019
Information Review	3 PM, Friday 6th of December 2019.
Provisional contents page	3 PM, Friday 21st of February 2020.
Draft critical evaluation	3 PM, Friday 27th of March 2020.
Draft report	3 PM, Friday 27th of March 2020.
Submit the project report to Turnitin	3 PM, Wednesday 22nd of April 2020.
Submit the project report, physical and electronical and copies of the deliverable	3 PM, Thursday 23rd of April 2020.
Demonstration of work	Before 3PM, Tuesday 12th of May 2020.

BCS Code of Conduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.
Signature: B.Moore

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.
Signature: B.Moore

GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire or participant testing materials. This form is available on the Bb site for the module.
Signature: B.Moore

Ethics

Complete the SHUREC 7 (research ethics checklist for students) form below. If you think that your project may include ethical issues that need resolving (working with vulnerable people, testing procedures etc.) then discuss this with your supervisor as soon as possible and comment further here.
Both you and your supervisor need to sign the completed SHUREC 7 form.
Please contact the project co-ordinator if further advice is needed.

RESEARCH ETHICS CHECKLIST FOR STUDENTS (SHUREC 7)

This form is designed to help students and their supervisors to complete an ethical scrutiny of proposed research. The SHU Research Ethics Policy should be consulted before completing the form.

Answering the questions below will help you decide whether your proposed research requires ethical review by a Designated Research Ethics Working Group.

The final responsibility for ensuring that ethical research practices are followed rests with the supervisor for student research.

Note that students and staff are responsible for making suitable arrangements for keeping data secure and, if relevant, for keeping the identity of participants anonymous. They are also responsible for following SHU guidelines about data encryption and research data management.

The form also enables the University and Faculty to keep a record confirming that research conducted has been subjected to ethical scrutiny.

For student projects, the form may be completed by the student and the supervisor and/or module leader (as applicable). In all cases, it should be counter-signed by the supervisor and/or module leader and kept as a record showing that ethical scrutiny has occurred. Students should retain a copy for inclusion in their research projects, and staff should keep a copy in the student file.

Please note if it may be necessary to conduct a health and safety risk assessment for the proposed research. Further information can be obtained from the Faculty Safety Co-Ordinator.

General Details

Name of student	Benjamin Neil Moore
SHU email address	b5024749@my.shu.ac.uk
Course or qualification (student)	Bsc Computer science for games
Name of supervisor	Jonathan Saunders
email address	acesjms@exchange.shu.ac.uk
Title of proposed research	**Game-based AI pathfinding implementation and optimisation**
Proposed start date	24/10/2019
Proposed end date	23/04/2020
Brief outline of research to include, rationale & aims (250-500 words).	Many genres of games use AI pathfinding. One of the most common genres you will see this implemented in is real time strategy games, where you would have units that you would want to get from point A to point B in the shortest time possible. But how does this work and how is it done? This project is to look at AI pathfinding more in depth, investigate the various techniques and understand how it works. It will include the implementation of both basic and advanced AI pathfinding algorithms, optimising the algorithm to improve both the efficiency and

	performance of the AI pathfinding. The prototype will allow multiple units to traverse through obstacles, finding the shortest path to the end destination. It will have real-time collision avoidance, when its path is blocked in real-time it will re-calculate a new path. By completing this project, I will: • Increase my knowledge of games related AI pathfinding. • Understand how to improve the efficiency and performance of the AI pathfinding. • Apply the knowledge from this project into other fields such as robotics. The project aims to achieve the following: • Research Game engines & Learn how to use chosen game engine. • Research and identify multiple AI pathfinding algorithms. • Research crowd AI pathfinding • Create a basic grid based 3d map in chosen game engine • Implement basic AI pathfinding • Implement crowd AI pathfinding • Improve efficiency and performance of the AI pathfinding.
Where data is collected from individuals, outline the nature of data, details of anonymization, storage and disposal procedures if required (250-500 words).	

1. Health Related Research Involving the NHS or Social Care / Community Care or the Criminal Justice Service or with research participants unable to provide informed consent

Question	Yes/No
1. Does the research involve? • Patients recruited because of their past or present use of the NHS or Social Care • Relatives/carers of patients recruited because of their past or present use of the NHS or Social Care • Access to data, organs or other bodily material of past or present NHS patients • Foetal material and IVF involving NHS patients • The recently dead in NHS premises • Prisoners or others within the criminal justice system recruited for health-related research***** • Police, court officials, prisoners or others within the criminal justice system***** • Participants who are unable to provide informed consent due to their	No
2. Is this a research project as opposed to service evaluation or audit? *For NHS definitions please see the following website* http://www.hra.nhs.uk/documents/2013/09/defining-research.pdf	

If you have answered **YES** to questions **1 & 2** then you **must** seek the appropriate external approvals from the NHS, Social Care or the National Offender Management Service (NOMS) under their independent Research Governance schemes. Further information is provided below.

NHS https://www.myresearchproject.org.uk/Signin.aspx

***** All prison projects also need National Offender Management Service (NOMS) Approval and Governor's Approval and may need Ministry of Justice approval. Further guidance at:

http://www.hra.nhs.uk/research-community/applying-for-approvals/national-offender-management-service-noms/

NB FRECs provide Independent Scientific Review for NHS or SC research and initial scrutiny for ethics applications as required for university sponsorship of the research. Applicants can use the NHS pro-forma and submit this initially to their FREC.

2. Research with Human Participants

Question	Yes/No
Does the research involve human participants? This includes surveys, questionnaires, observing behavior etc.	Yes
Question	**Yes/No**
1. *Note If YES, then please answer questions 2 to 10* *If NO, please go to Section 3*	No
2. Will any of the participants be vulnerable? *Note: Vulnerable' people include children and young people, people with learning disabilities, people who may be limited by age or sickness, etc. See definition on website*	

3. Are drugs, placebos or other substances (e.g. food substances, vitamins) to be administered to the study participants or will the study involve invasive, intrusive or potentially harmful procedures of any kind?	
4. Will tissue samples (including blood) be obtained from participants?	
5. Is pain or more than mild discomfort likely to result from the study?	
6. Will the study involve prolonged or repetitive testing?	
7. Is there any reasonable and foreseeable risk of physical or emotional harm to any of the participants? *Note: Harm may be caused by distressing or intrusive interview questions, uncomfortable procedures involving the participant, invasion of privacy, topics relating to highly personal information, topics relating to illegal activity, etc.*	
8. Will anyone be taking part without giving their informed consent?	
9. Is it covert research? *Note: 'Covert research' refers to research that is conducted without the knowledge of participants.*	
10. Will the research output allow identification of any individual who has not given their express consent to be identified?	

If you answered **YES only** to question **1,** the checklist should be saved and any course procedures for submission followed. If you have answered **YES** to any of the other questions you are **required** to submit a SHUREC8A (or 8B) to the FREC. If you answered **YES** to question **8** and participants cannot provide informed consent due to their incapacity you must obtain the appropriate approvals from the NHS research governance system. Your supervisor will advise.

3. Research in Organisations

Question	**Yes/No**
1. Will the research involve working with/within an organization (e.g. school, business, charity, museum, government department, international agency, etc.)?	No
2. If you answered YES to question 1, do you have granted access to conduct the research? *If YES, students please show evidence to your supervisor. PI should retain safely.*	
3. If you answered NO to question 2, is it because: A. you have not yet asked B. you have asked and not yet received an answer C. you have asked and been refused access. *Note: You will only be able to start the research when you have been granted access.*	

4. Research with Products and Artefacts

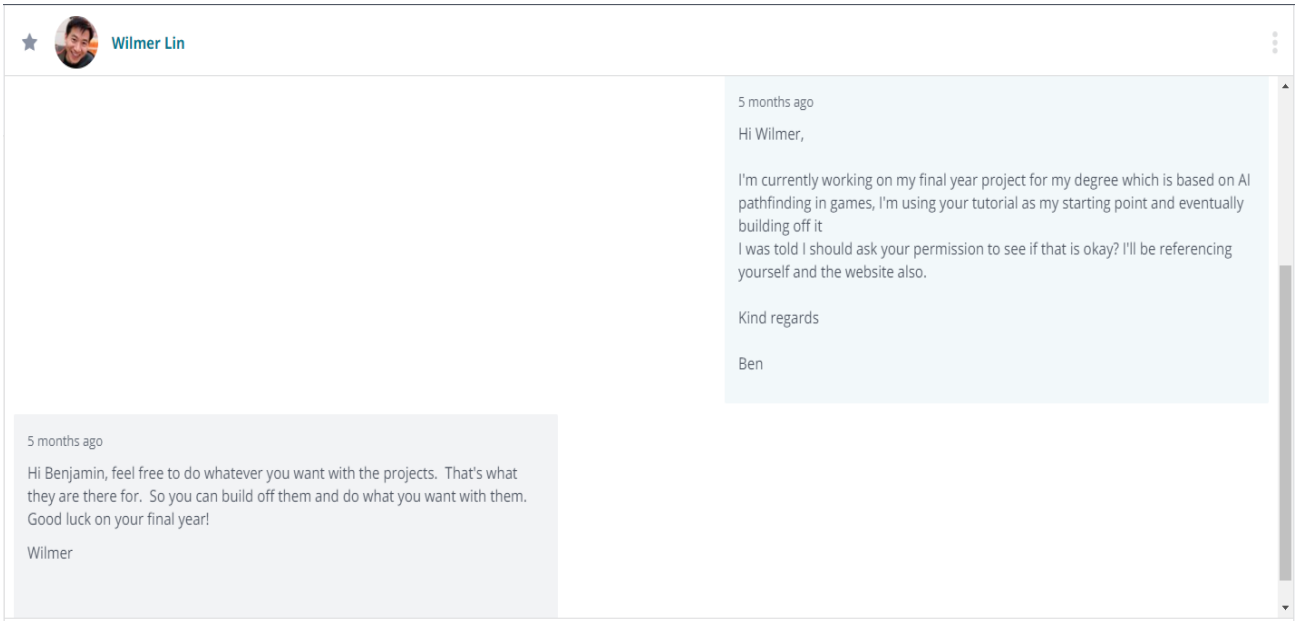
Question	**Yes/No**
1. Will the research involve working with copyrighted documents, films, broadcasts, photographs, artworks, designs, products, programmes, databases, networks, processes, existing datasets or secure data?	Yes

2. If you answered YES to question 1, are the materials you intend to use in the public domain? *Notes: 'In the public domain' does not mean the same thing as 'publicly accessible'.* • *Information which is 'in the public domain' is no longer protected by copyright (i.e. copyright has either expired or been waived) and can be used without permission.* • *Information which is 'publicly accessible' (e.g. TV broadcasts, websites, artworks, newspapers) is available for anyone to consult/view. It is still protected by copyright even if there is no copyright notice. In UK law, copyright protection is automatic and does not require a copyright statement, although it is always good practice to provide one. It is necessary to check the terms and conditions of use to find out exactly how the material may be reused etc.* *If you answered YES to question 1, be aware that you may need to consider other ethics codes. For example, when conducting Internet research, consult the code of the Association of Internet Researchers; for educational research, consult the Code of Ethics of the British*	Yes
3. If you answered NO to question 2, do you have explicit permission to use these materials as data? *If YES, please show evidence to your supervisor.*	
4. If you answered NO to question 3, is it because: A. you have not yet asked permission B. you have asked and not yet received and answer C. you have asked and been refused access.	**A/B/C**

Adherence to SHU policy and procedures

Personal statement	
I can confirm that: – I have read the Sheffield Hallam University Research Ethics Policy and Procedures	
Student	
Name: Benjamin Neil Moore	Date: 24/10/2019
Signature: B.Moore	
Supervisor or other person giving ethical sign-off	
I can confirm that completion of this form has not identified the need for ethical approval by the FREC or an NHS, Social Care or other external REC. The research will not commence until any approvals required under Sections 3 & 4 have been received.	
Name: Joe Saunders	Date: 25/10/2019
Signature: Joe Saunders	

Appendix I - Permission from Wilmer Lin.



Figure[124]: Permission from Wilmer Lin.