Pathfinding for Multiple Agents

BSc Computer Science

Adam Simpson(B9031265)

Project Supervisor Dr Giacomo Bergami

Word Count: 12,094

Abstract

This dissertation investigates some of the algorithms and heuristics that could be used in the gaming industry for the pathfinding of agents. Agents in this case are single objects that move from one point to another, that resemble enemy AI (Artificial Intelligence) for a game. It compares the performance and speed of the algorithms and heuristics, to give a full understanding of which could be considered the best combination. It concludes that a Manhattan heuristic with an A* algorithm that is called only when an obstacle is blocking the path was the best combination

The driving motivation for doing this dissertation project is an interest in video games and the industry as whole. While pathfinding is only a small subsection of the industry it allows for a way of giving back to an industry which has given such joy to many people by giving the developers the results, they would need to decide about which algorithm or heuristic is best for them.

Declaration

“I declare that this dissertation represents my own work except where otherwise stated.”

Acknowledgements

I would like to thank my supervisor Giacomo Bergami for guiding me throughout the project. I would also like to thank my friends and family for supporting me during the lower times throughout the project.

Table of Contents

Table of Contents

Table of Figures

1 Introduction

1.1 Context and Motivation

1.1.1 The Context

Path-finding algorithms are the backbone of AI in the gaming industry. Games such as the Orcs Must Die series use pathfinding for every single moving enemy AI. The enemies in Orcs Must Die constantly have to change path due to barricades the player can put down, which stop the enemies from moving in through that point. Path-finding algorithms are a required part of any AI that is not stationary, whether that be simply patrolling a single line, chasing the player, or following more advanced rules such as using Goal Oriented Action Planning (GOAP). The more advanced rules such as GOAP can be used to path find when actions and operations are associated with the costs of performing an action, in path finding's case, moving to specific areas.

The current state of the art algorithms used in the games industry vary depending on development team or tool used. Some that do not require much complexity use Djikstra since it is the easiest to implement, while others use the Navigation Mesh (NavMesh) in Unity because it is in-built into Unity. However, current state of the art algorithms used in general are A* based, with additions or improvements being used to make them complete a particular task faster, or not use up as much memory on exploring a part that has no need to be explored, like in the case with the AAAI algorithms "Dead-End" and "Gate-Way"(Björnsson and Halldórsson, 2006). There are a lot of different path-finding algorithms, and it can be hard to determine which is best, or if the algorithms currently used have inexpensive improvements that have been discovered but have not become common place in the games industry. This knowledge would be useful to many game developers across the world who are doing pathfinding. It will be helpful because knowing which algorithm is suitable to their game can take a long time, time that could be spent on other parts of development. The games industry already has a reputation of putting their employees under immense time constraints and lessening the pressure even on just picking an algorithm could make a difference to how much a game is finished or how polished it is.

1.1.2 The Problem

In the current state of game development, many path-finding algorithms are used, with A* being the most common. There are others which are company or game engine specific which are harder to know or test. The problem is many developers either do not know about other path-finding algorithms, or do not know the benefits they could bring. Therefore, seeing which path-finding algorithms is the fastest and/or most efficient could be useful to developers. However, most games do not just have one agent moving but instead multiple agents. This complicates things more and means pathing algorithms have another layer of complexity. Therefore, adding in multiple agents means a developer can see if that will make a difference to the algorithms and if some algorithms are better at single agent or multiple agents. This project will not help those companies who use a specific algorithm to do pathfinding but will help those who are either independent or have flexibility with what algorithms they can use.

1.1.3 The Rationale

The two main pieces of data that are needed and useful to developers are, how fast is the algorithm in completing and how much memory does it take while running or how much it slows down when running. Therefore, I intend to study this data by creating multiple maps, increasing in complexity, and testing them first to see if the algorithms can finish the maps. I will then collect the data such as

how long it takes to finish and how much memory is used. I will also make notes on how slow it runs and comment on it at the end of the project. The data will be stored in a table and will be done multiple times to give an average. This data is important because developers could look at the data and it could really help them decide which algorithm to use. I aim for this project to be as useful as possible to developers, while exploring in as much detail the algorithms proposed in the background section.

1.2 Aims and Objectives

The aim of the project was to:

- Analyse the performances of different pathing algorithms to determine which is best suited to a group of agents or AI in the gaming industry.

The aim was then split into these objectives:

1. Research and identify 2 aspects of pathing algorithms which can be compared to determine a ranking of A* along with other pathing algorithms. These became time taken to complete the maze and average fps during running the maze which can be seen in the results in Figure 5 and 6.
2. Design and implement an A* algorithm and test the algorithm to make sure it performs path-finding tasks adequately.
3. Analyse the performances of the A* algorithm using different heuristics.
4. Implement the improvements found from the research and test to make sure they perform path-finding tasks adequately.
5. Analyse the performances of the improvements using the same heuristics as before.
6. Implement any other Algorithms found in future research and analyse against previous algorithms.

The two aspects identified for objective 1 became time taken to complete the maze and average fps during running the maze which can be seen in the results in Figure 5 and 6. Objective 2 and 3 can also be seen in the results section. Objectives 4, 5 and 6 would become future work in some respect.

1.3 Structure of the Dissertation

This dissertation will start by going into the technical background for the dissertation including the research that happened before the main body of work began. It will then go into the technologies that were used for the main body of work. After that, this dissertation will go into the design and implementation of the work, including the testing strategies that were implemented as the work progressed. Then the most important part will be introduced, which is the results and evaluation part of this dissertation. The results will hope to show which algorithm can be considered the best under certain criteria and these criteria will also be evaluated to see if it could be improved upon. Finally, the dissertation will move on conclusions. This will involve comparing the start of the project to now, to see what changed and why. It will also include what has been learnt over the project and how this could be beneficial to the future, which leads into the final part of the section which is the future work. The future work section will go into what could be done in the future to improve upon or expand the work completed during this project.

2 Technical Background

2.1 Research

2.1.1 Video Games

Many video games use pathfinding to plan out a route for agents to move across the map. Pathfinding is not restricted to one type of video game; it can be seen in real time or turn based strategy games or in first person shooters. This can also take many forms, from helping a player traverse their units, or having enemy agents plan how they will traverse a map. One example is in strategy games like the Civilisation series, where both the player and AI use pathfinding to plan a path to traverse their units across the map. In the case of Civilisation, hexagonal nodes are used, meaning the heuristic for pathfinding can use the six directions. Another example of pathfinding in video games is within the Orcs Must Die series.

Orcs Must Die is a video game series that mixes tower defence and third person combat where the player must protect a “rift” which is in other terms the end point for the enemies.(Entertainment, 2012). The enemies use pathfinding to get to the end point which deducts “rift points” from the player and if the points reach 0 then the player loses. The reason why this is relevant to the project is because of the pathfinding. At first, something similar was to be attempted but then there was a realisation that this might be too complicated in the time given. However, it is still a very prevalent example of pathfinding in video games. It is also a very credible source due to how well received it was by the gaming community for the first game. Metacritic for example gave it high marks for both PC(Metacritic, 2011a) and Xbox(Metacritic, 2011b) releases. However, it also received high marks by Metacritic for the most recent game Orcs Must Die 3.(Metacritic, 2021)

The Orcs Must Die series has at points hundreds of enemies and, potentially up to a thousand enemies in the latest game due the new “War Scenarios”(Ide, 2020). Therefore, the algorithm used by the developers to keep the game running at a smooth pace with so many enemies at once must be extremely well thought out. The enemies can also attack the player if the player happens to get too close. Having up to a thousand enemies path finding at once, while checking if the player is in range to hit is impressive but the player can also place barricades on parts of the map during runtime, stopping enemies going that way. The enemies once they get to those barricades then change their path because they realise the way they were going is blocked. If all ways to the rift are blocked, then the enemies recognise this and will destroy a barricade so they can get through. From experience Orcs Must Die 3 sometimes suffers from the enemies going to one barricade, realising they cannot go through, changing path, going along that path, and then realising that is also blocked. Then the enemies may go back to the first barricaded area, causing them to either get stuck going back and forth or finally working out the correct way around. This however was very rare when experiencing the game and only happened in a couple of instances when the level had many enemies working at once in enclosed areas.

The pathing algorithm that this video game has works very well for what it is attempting to achieve. It awards the player for smart barricade placements that slow the enemies down. However, the issue that this algorithm would have if it were implemented in other settings would be that it does not take barricades into account until each agent reaches it. If a single agent reached it and then all agents following a similar path, or with that node in common, knew not to take that path, this algorithm would be much more applicable. This means finding an algorithm which could eliminate the area in an efficient way from the pathfinding algorithm would be appropriate to somewhat improve the algorithm. This is where “Zoning” algorithms have an advantage.

2.1.2 Zoning Algorithms

There were a few differing improvements to A* found in AAAI. Some of them were useful, while others were not due to various factors.

The paper that was chosen from AAAI (Björnsson and Halldórsson, 2006) was thought to be the best choice. The algorithms from the paper became a technology that was to be implemented. These were two pathfinding algorithms and one decomposition algorithm. The decomposition algorithm was used as pre-processing for the two pathfinding algorithms. The decomposition algorithm sectioned the map up into zones dependant on a set of rules established by the paper. The first pathfinding algorithm, the “dead-end” algorithm, knows whether the room that has been entered is a dead end. A dead end is explained as “there is no pathway from this room to the goal (except back out via the entrances we came in through).”(Björnsson and Halldórsson, 2006). The other algorithm is the “Gateway” algorithm. This involves pre-calculating during the pre-processing phase the distances between all entrances and exits of every area. Then during the runtime phase using a heuristic to calculate the distance from a cell to a gate.

The paper seemed useful as it could be done on a grid environment and had visual aids so that a developer could see what was being used. It also included pseudocode for the decomposition algorithm which made it easier to implement than attempting to code from scratch. It would have been even more useful if the pathfinding algorithms also had pseudocode, but there are very good descriptions of how they work and perform. One of the reasons why this algorithm was chosen was because some maps from the game Baldur’s Gate II, a role-playing game from 2000, had already been experimented on. Another reason was that the paper had suggested there was further work to be done by attempting it on more maps. Therefore, by creating the algorithm and testing it on a new map, it would be following their future work. However, due to time constraints, only the decomposition algorithm was completed, meaning the two pathfinding algorithms were not completed.

2.1.3 Obstacle Avoidance Algorithms

There were many differing improvements to A* in IEEE Xplore. These ranged in usefulness due to the ways in which they approached their papers.

For example, a paper from IEEE (Ju;Luo and Yan, 2020), discussed an improvement to A* that involved a line getting from one corner to another, using obstacle avoidance. This paper showed that the improved A* would attempt to go back to the straight line from corner to corner, instead of forming a parallel line that would eventually change to reach the corner. The improved A* showed an effective difference in pathing, reducing the path by almost 1 metre when compared to traditional A*. However, this was at a sacrifice of an almost 30 second increase.

While times over 40 seconds may be useful in some respects, it was not made for gaming where calculation time needs to be quite quick. A player would not be very happy especially with today’s technology if it took over 40 seconds to calculate a path. There is potential that the time of over 40 seconds could be reduced significantly on a very high-end PC and therefore could be viable for games on a very high-end PC. However, a console would never be able to reduce the time by a significant amount to make this a viable solution. Therefore, this meant that this paper was not as relevant as first believed and a decision was made that the algorithm in this paper was not to be considered due to how long the calculation time would take.

2.2 Technology

2.2.1 Unity

Unity was decided to be the main technology used for two main reasons. The first reason was that for it to be in a game like environment, the visual aspect was required since all games have a visual component. The two main environments for this would be Unity or Unreal. These engines allow for easy ways of contrasting a visual component that does not take much time to create. It also allows for an easier understanding of what is happening since it is not just pieces of text, such as debugs, but a visual piece, where it can be seen something is working.

The second reason was because Unity had been used before on other projects so therefore there was some experience prior to starting that could have been used. The other option would have been Unreal however, this was in a new language, C++, and a new environment. This would have meant learning both the language and the environment which would have taken more time than just using Unity with the language C#.

2.2.2 Pathfinding Algorithms and Heuristics

Among all the possible pathfinding algorithms, few of them are often practically used. Dijkstra is one of the most common pathfinding algorithms and is seen as the simplest graph-based pathfinding algorithm(Flam, 2018). A* is a variant of the Dijkstra algorithm that is most used in video game development and is seen as superior to Dijkstra because it is said that the A* algorithm is a “provably optimal solution for pathfinding” (Cui and Shi, 2010). Another pathfinder is using an A* algorithm with the Unity NavMesh. NavMesh is an abstract data structure that is used alongside A* for pathfinding in Unity only. The abstract data structure means that Unity takes a lot of the work on making it work. This felt disingenuous to use instead of using an algorithm created during the project. However, in hindsight if this was used, it would have meant more time could have been spent on parts of the project there were incomplete, but it would have meant those in other engines could not have used this paper. To showcase a solution that might also be extended to other game engines, we discarded NavMesh, as this solution cannot be externalised to other game engines. Therefore, after reviewing the other pathfinding algorithms, it was decided that A* would be chosen due to how used it is in the games industry already and because it was already considered to be a variant of Dijkstra that was better.

A heuristic is defined as a strategy that uses already available information to control the processes of solving a problem (Pearl J, 1984) or is also defined as a process or method (Lexico, 2022). One heuristic in the case of this project is an approach to prune the set of all possible strategies by approximating additional information that might be used to solve the problem. The overall aim is to find an acceptable approximation to the solution while decreasing the overall computation time required for visiting the exact solution.

The main heuristic distances that were to be investigated were using Manhattan distance and Euclidean distance (sometimes referred to as Octile distance due to it using eight directions for movement). These are the most common heuristics for pathfinding which made them key to be investigated. Euclidean is calculated using Minkowski’s distance formula, which is the difference between two points in a 2D space. Manhattan on the other hand is “calculated using an absolute sum of difference between cartesian co-ordinates”(i2tutorials, 2019). Another definition is the total number of North, South, East, and West operations required to reach a given destination. There was a consideration of using Hamming distance, but this was decided to be not as important because Euclidean and Manhattan are more commonly used.

3 What was done, and how

3.1 Design

3.1.1 Design Introduction

The maze was carefully designed to include many situations that would be encountered in a maze, while also keeping it small enough so that the computer did not have a problem storing all the different nodes and variables for each node. In the end the number of nodes came to about 1750 which was more than was expected for what was thought of as a small map. Figure 1 shows the maze that was used to take the results that are in section 4.1.

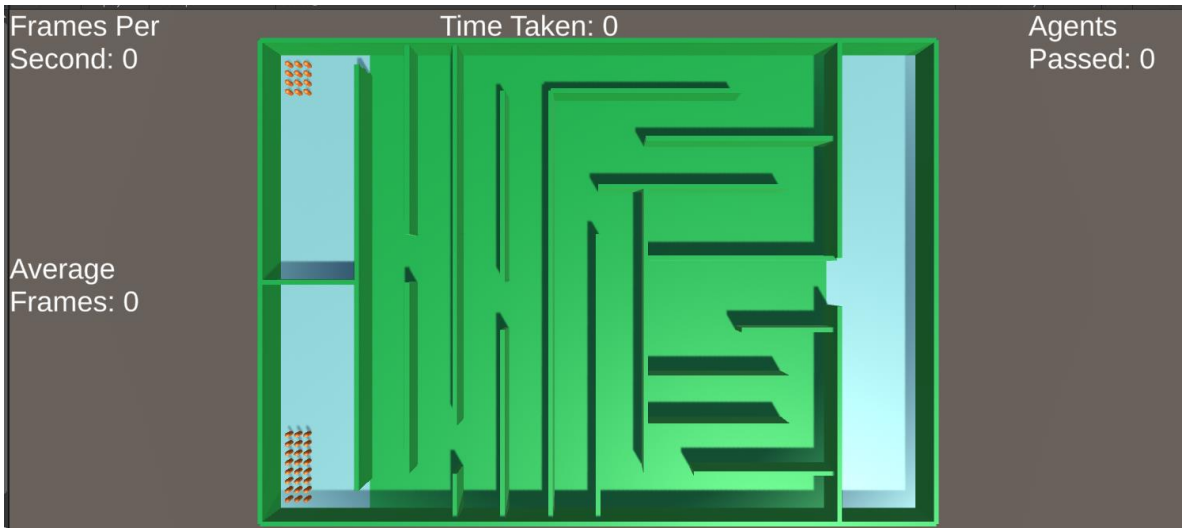

Figure 1 – Picture of the maze, before pathfinding and therefore movement begins.

3.1.2 Path of the Agent

The path of the agents was carefully designed so that it was easy to see, for a person looking at the maze, that the agents were taking the right path. This included putting dead ends in places that were clearly dead ends and did not loop around. The dead ends were to ensure the algorithm was working and because most mazes have dead ends, especially in games. Games such as the Orcs Must Die franchise have dead ends in a different way where the player places “barricades” and the enemies must work their way around. This felt like it would be too complicated to incorporate in the time given but the shape of the maze in figure 1 could be like a route the enemies might take in Orcs Must Die. Dead ends can also be used in a different way in Orcs Must Die, where it can be used to trick enemies into following a long path, to only go all the way back and follow another way to the destination. This in the game makes sense as a gameplay feature but it was a feature that was decided not to be implemented in the maze in figure 1. A* does the pathing to avoid going back, so the Orcs Must Die developers must use a different algorithm for their game. Therefore, while there

could not be an attempt to make something similar and investigate how it works and performs, this influenced how the maze was created.

Figure 2 – Orcs Must Die Picture

– Picture of Orcs Must Die, shows barricades being used to funnel enemies down a single area.(ZonaFree2Play, 2011)

There were also some places where there was more than one way to go to see whether the agents would take the fastest route. This was done purposefully, and the difference was large enough so that it could be seen by a human looking at the agents going through the maze.

The decision was also made to separate the agents, to see how much the algorithms would affect agents at different areas of the map. On accident this is also like Orcs Must Die as there are some maps where enemies will come out of different gates, however they normally are on opposite sides of the map and not on the same side. Another reason to separate the agents was to see what happened if they ever interacted with each other, would they collide and cause issues or would they be able to handle the new information and move in accordance with the rules that were set in the algorithm.

3.1.2 Rule Design for algorithms

Algorithm 1 and Algorithm 2 have distinct differences that make them have different behaviours. As the agent moves with each update the 2 algorithms do different strategies. Algorithm 2 calls the pathfinding algorithm on each update, so it is constantly renewing the path it is taking. Algorithm 1 on the other hand only renews the path and calls the pathfinding algorithm when it runs into either criterion. The criteria are, if the next node has an agent on it or if the next node has already been "claimed" by another agent. If a node has been claimed, then that means the agent who claimed it will be moving onto that space on the next update. To stop the agent, that is waiting, from calling the pathfinding algorithm multiple times while the agent, that claimed the node and is moving, moves onto the node and then moves away from the node that was claimed, a counter was put in place. This counter waits 4 updates, to make sure that the agent is not still on the node in front and then calls the pathfinding algorithm to start from where the agent is and finish where the goal position is. This stops multiple calls to the pathfinding algorithm which makes it more efficient.

3.2 Implementation

For someone to reproduce the results, a way of accessing the work has been devised. Due to issues with getting a GitHub repo working, a Google Drive link has been used instead. The link for this Google Drive is this:

https://drive.google.com/file/d/1gtoibVStB-NBOugnIY-Aoiphek6WTuTX/view?usp=sharing

3.2.1 AgentMovement

AgentMovement has the prime function of moving each agent individually to the next node that has been specified in PathFinding. It also deals with destroying the object once they reach their destination. Furthermore, it deals with issues where two agents are trying to move to the same node, or an agent is trying to move to a node which already has another agent on the node.

3.2.2 Grid

Grid does a lot of different jobs that are connected to each other. First it partitions the maze into x and y coordinates by discretizing the environment into a block of a fixed length, thus portioning the whole grid into distinct blocks. This is key to know because a larger grid size will mean there are more nodes, and the more nodes will be reflected into an increase of the required running time for the algorithm, as the total number of nodes and edges to be visited will increase.

Then the grid is created using a node matrix, where each dimension reflects one of the two axes. This means that each node is accessed in constant time when the node coordinates are known. Each node in the grid created is then associated with variables of relevance to the algorithm.

The same class also implements the AAAI algorithm: the first decomposition step sets each node to a zone. Every node in the grid that is not a wall and therefore is “free”, must be associated to a zone. Zones are a cluster of merged nodes that identify contiguous regions belonging to the same rooms, thus breaking down the grid with a greater granularity. The subsequent steps linking “Gateway” and “Dead-End” from the paper was not implemented due to time constraints and issues with getting other parts working. However, since the decomposition algorithm was completed, it was included as I might draw important lessons by attempting to implement this algorithm; these are going to be discussed later in the dissertation. The pseudocode does not clarify some data structures, such as a sorted set, might be required for effectively implementing the pseudocode. This allows to keep track of all the “free” nodes in the grid. The decomposition algorithm would use such a set to get the next “free” node by removing nodes that were assigned to a zone and obtaining the first position in the set, which would be the next free node.

3.2.3 SetSorter

SetSorter uses a set of key value pairs to store the x and y component for each node on the grid. We need to sort these coordinates to retrieve the minimum node grid being free, and to ensure that each coordinate is available only once. Sets enforce the coordinates’ uniqueness by exploiting red-black tree and by accepting an ordering predicate; in our scenario, we exploited the usual lexicographical order to sort the grid nodes by first comparing the x axis and then y axis.

3.2.4 Node

Node is used to create a “Node” object. This object was required to encapsulate fields required by the algorithm by exploiting the responsibility principal design pattern: this ensures that all the properties of the node should be stored in one single object. The Node object is crucial as it might be

exploited to retrieve its location in the world space. It could also state whether the Node was part of a wall or if an agent was on it or if it was “free” to move to.

3.2.5 PathFinding

PathFinding consists of all the maths to do with path finding. This includes the gcost, hcost and fcost. The gcost or g(n) is the cost from the start node to a specific node. The hcost or h(n) is the estimated cheapest cost from a given node to the end node. The costs are standard use for A* algorithms since Bertram Raphael suggested them in circa 1966 – 1968(Nilsson, 2010). The fcost, f(n) is equal to the gcost plus the hcost (g(n) + h(n)) and is therefore the total cost.

A list is used to store each node which is called an “OpenList”. The list is used to store nodes so that they can be tested under certain criteria to see if the node is the next node that should be travelled to. If the node meets the criteria, then it is added to a HashSet which is then compared to its neighbours to check whether the neighbours are walls, or if the fcost is less than the gcost of the neighbouring nodes. Once there are no more nodes in the “OpenList” and the final or target node is the same as the current node being inspected, the final path is then calculated.

The final path adds each node to a list starting from the final node and works back towards the start node. Once the start node is reached, it is added to the list and the list is reversed to make the list from start to end, giving the agents a path to follow.

The Manhattan distance is also calculated in this file. The Manhattan distance is a way of estimating the distance between a specific node and the final node. This entails finding the difference of each component between a given node and the final node. When the difference of both components is added together, this is considered the Manhattan Distance. The distance which has been estimated is called h(n).

3.2.6 TextManager

TextManager works with the canvas in Unity to display any text such as the average fps, the current fps, the time taken, and how many agents have passed through the maze. The canvas in Unity is a way of displaying text on screen that the player can see, which do not affect the game world underneath, but the text can be affected by what happens in the game world. The single responsibility principle helps at coalescing all the relevant text information in one single class. This choice resulted in a time performance gain as files required to display the time taken, fps and agents passed were done only once, not on multiple separate occasions.

3.3 Testing Strategies

Testing for this project happened as the project progressed. The reason for this was because the development was of a more agile type, meaning it did not use the more typical models such as waterfall or spiral models. This is further discussed in the “Evaluation of the Plan” section. However, due to the project being more agile, it meant that testing it had to be too. This meant that as the project progressed there needed to be ways to quickly check whether there were bugs and how to solve them. The two main ways that were extremely useful in testing the project was the Unity’s visuals, including using Gizmos and the use of debugs, especially when running in to the most common of errors in Unity and C#, the NullReferenceException, which usually means that a variable that is trying to be called has no value.

3.3.1 Unity's Gizmos

Unity has a visual debugging tool called "gizmos". Gizmos are most used to highlight and draw shapes to be used as a way of debugging visually (Unity, 2022). In the case of this project, it was used to draw cubes to represent each node in the grid. As seen below in figures 4 and 5, there is a red line that starts from the left side of the screen and works its way around to the right. This red line is the fastest path perceived by the path finding algorithm. Figure 3 shows an example path of an agent starting in the bottom part of the left side, while figure 4 shows an example path of an agent starting on the top of the left side. Gizmos were a testing strategy because of how useful they were in working out what was going wrong visually. In the early stages this was used much more because I could figure out if the line seemed to be the fastest from just a glance. It also helped showing which nodes would be affected by the walls. The walls which are green in the figures below, have an aura around it which are nodes that are counted as being a part of the wall, these are in yellow. This meant that if the yellow and red overlapped it was obvious that something in the code was not working as intended due to having a visual representation of it.

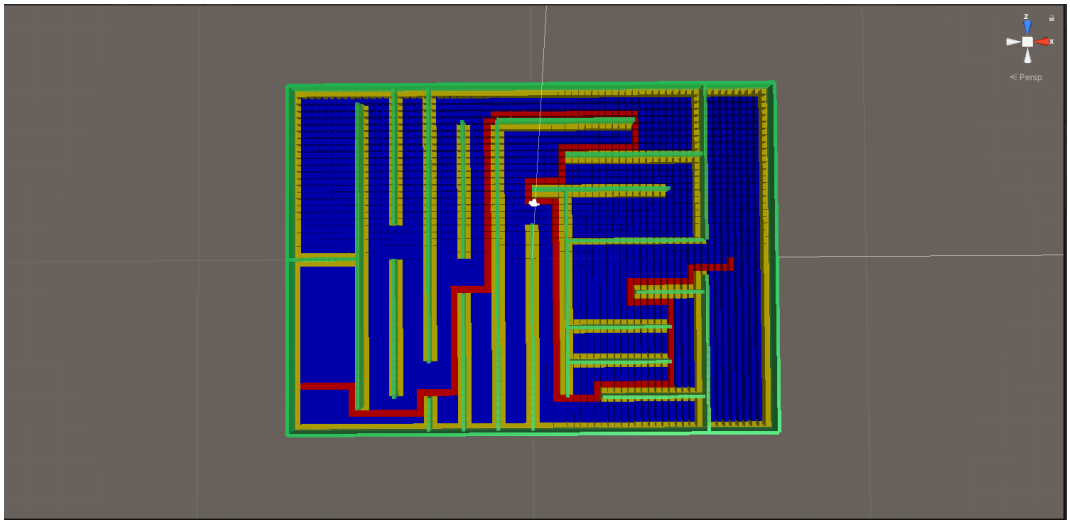

Figure 3 – Picture of maze, showing fastes path (shown in red) for agents in bottom area.

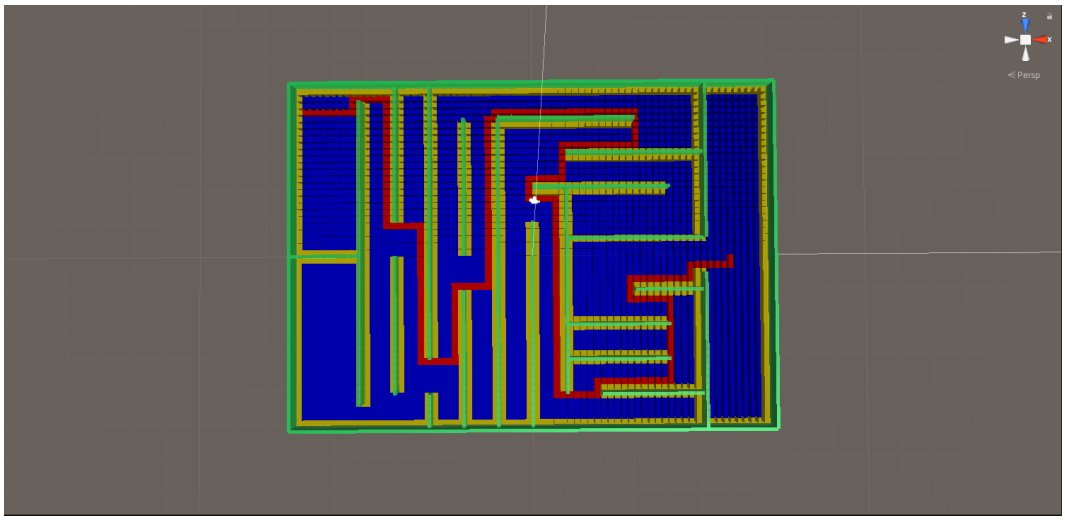

Figure 4 – Picture of maze, showing fastest path(shown in red) for agents in top area.

The blue shows nodes that were not currently being used for the path and these were “free” nodes. This was then changed for the decomposition algorithm. Due to the decomposition algorithm sectioning the grid into zones, a visual aid was used to see if the sectioning was happening correctly.

3.3.2 Use of Debugs

The use of debugs was a very important part of testing. However, due to the nature of debugs only giving what the programmer asks for, it takes some skill to use them effectively. This skill improved as the project developed. The way debugs were used in this project were mainly when an error would occur such as the “NullReferenceException”. A debug would be used to check the variables in the line where the error occurred to see if any of them were null, if they were then the problem would be identified and then could be solved. If the debug did not help, then it would be expanded to variables that were connected to the variables in the line. For example, if the node object itself was not the issue, I would explore the variables that make up the node object checking to see if they were null when they should not be.

Debugs also helped when used in connection with gizmos. This is because the visual aid of the gizmos could give a very good idea of what part had gone wrong but not necessarily the exact problem. Debugs could then be used to isolate the issue and fix it similarly to how debugs were used when an error would occur.

3.3.3 Use of a Test Map

A test map was used more for the earlier stages of the project to make sure that the algorithm was working and that agents were moving properly through the environment. This was an important strategy because it allowed for fast testing on a smaller map, where it was obvious if an error had occurred. The testing of gizmos was also done here, to ensure that when it came to the map used for the results that they were working as expected. The reason the test map was used rather than using the results map was because if the results map was tweaked it could cause other issues, while if the test map was tweaked, it did not matter because it was only used for testing one thing at a time at first. Eventually it evolved to be testing how each part would work together but started as a way of testing very specific parts like the gizmos example.

4 Results and Evaluation

In this section, we will observe the results and what they suggest, regardless of context at first. Then in the evaluation section, the context will be given and explained, with comments to suggest how they could be improved if the tests were performed again.

4.1 Results of the Project

The results of the project are focussed on two similar algorithms with a difference that makes a considerable impact. It is also separated by which heuristic is being used as this would greatly skew results if they were mixed. The results of the project will be split into two parts. The first is the overall time taken versus how many agents there were. The second is the frames per second (fps) versus the number of agents which measures the responsiveness of the algorithms, an important part of games as the algorithms must be responsive to run smoothly.

4.1.1 Time Taken VS Number of Agents, Euclidean and Manhattan

Figure 5 seen below shows that algorithm one is always slower than algorithm two, no matter the number of agents. However, what is good to see is that they both follow the same trend, except for the end. Figure 5 also shows that Euclidean takes longer, the more agents there are. However, when there is only one agent, Euclidean is faster by an extremely small amount when comparing algorithm 1. However, when comparing algorithm 2 Euclidean is not faster than Manhattan. The difference on the other hand is so miniscule it does not show up on the graph at this resolution.

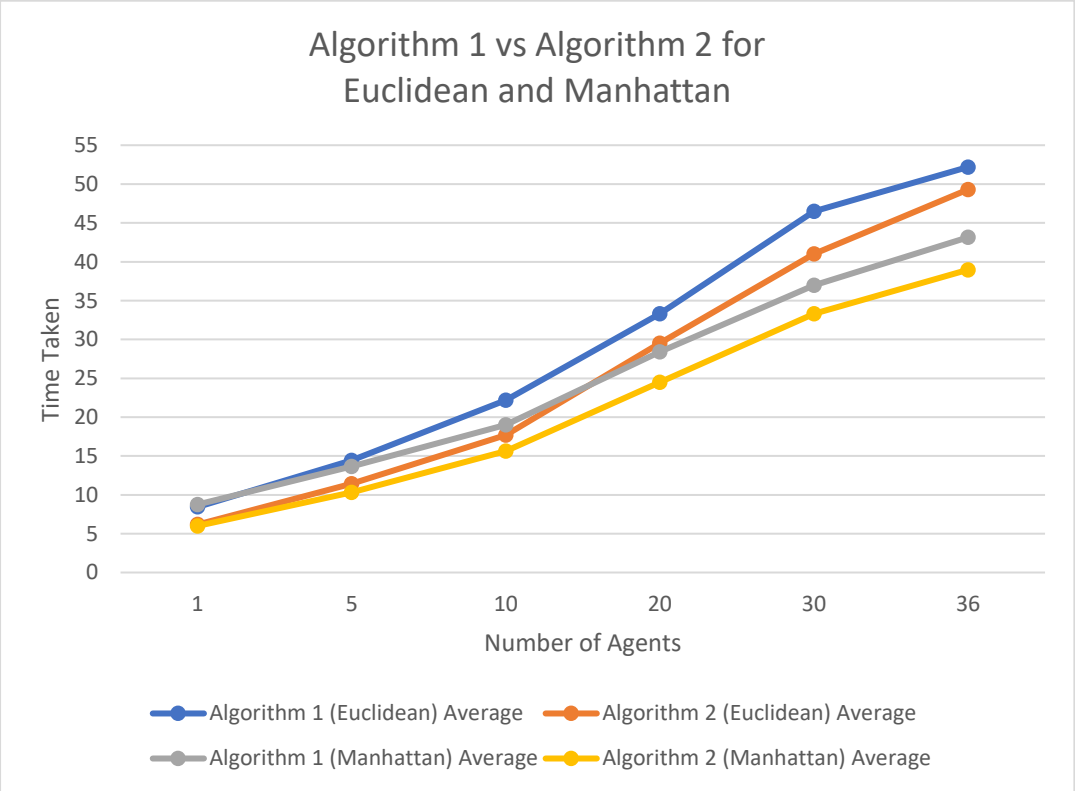

Figure 5 – Time Taken vs Number of Agents

4.1.2 Frames per Second VS Number of Agents, Euclidean and Manhattan

Figure 6 shows very clearly that algorithm 1 keeps a higher average fps than algorithm 2 when it comes to using a Euclidean Heuristic. Figure 6 also shows a comparison of the heuristic's Euclidean vs Manhattan for algorithm 1 and shows that the Manhattan average fps is higher in all cases than Euclidean. For algorithm 2 Manhattan has a higher average fps in most cases than Euclidean which is a similar trend to algorithm 1.

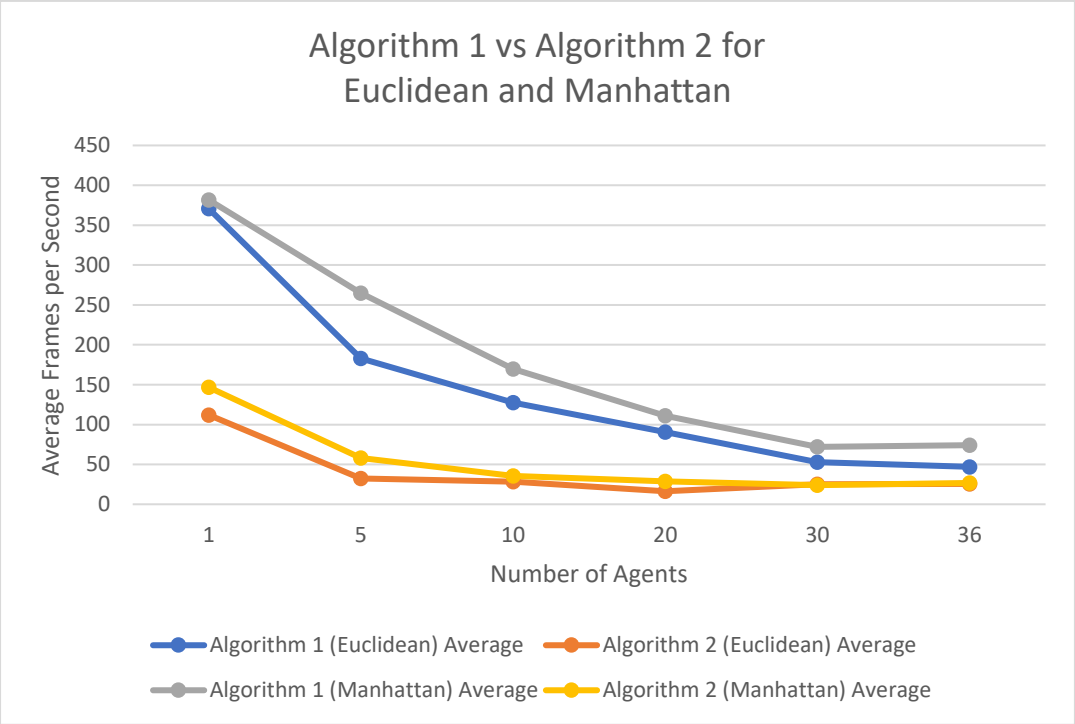

Figure 6 – Average FPS vs Number of Agents

4.2 Evaluation of the Results

4.2.1 Figure 5 – Time Taken Results

Figure 5 clearly shows a difference of 3 to 5 seconds between algorithm 1 and algorithm 2 for Euclidean. Figure 5 also shows a similar difference for Manhattan. The original prediction was that this would be the opposite and that algorithm 1 would be faster than algorithm 2. This was proven wrong by the data, but the real question is why was this the case?

The believed reason this was the case is because even though the second algorithm relies on running A* on each run, it keeps every agent moving as fast as possible. The first algorithm relied on a counter that would be set in motion whenever the space in front was blocked or had been claimed by another agent to stop from two agents trying to move onto the same place at the same time. The reason a counter was used was to stop the agent from immediately attempting to move onto the position that was blocked or claimed. Therefore, while this counter helped stop agents colliding, it made the time for all agents to complete the maze take longer.

However, this does not explain why for one agent this case still holds. Algorithm 1 should be faster in that one case due to there being no other agents to get in the way. This would mean that there is something else affecting the results that are unknown, or that a very unexpected outcome happened during the test and in fact this is the correct result.

Figure 5 also clearly shows that Euclidean takes longer, when using algorithm 1, the more agents there are. However, when there is only one agent, Euclidean is slightly faster. This would be hard to see by a person if they were to watch them side by side, due to such a small difference. The original prediction is that the Euclidean heuristic would be faster due to requiring less nodes to travel than the Manhattan heuristic.

However, the reason this is not the case is that each agent must complete an A* algorithm before needing to move, meaning more nodes will be explored by the Euclidean than the Manhattan. This is also true for every time an agent needs to stop to let another pass. Therefore, the larger number of agents, the more possible number of times where an agent needs to stop to let another pass or cannot move forward, the more calls are made to the algorithm. The more calls made means the greater number of nodes explored in total, which could slow down how fast the agents can complete the maze.

Euclidean being slightly faster at one agent follows the prediction but every point after that does not. The results clearly show that the opposite is in fact true, and that Manhattan is faster.

4.2.2 Figure 6 – Average FPS Results

Figures 6 shows a trend of algorithm 1 having a higher average fps than algorithm 2 at all numbers of agents. The original prediction was that algorithm one would have a higher fps, even higher than what it is at the more numerous amounts of agents. However, there are inconsistencies where there is a small increase in fps for algorithm 2 (Euclidean) at 30 and 36 agents and for algorithm 2 (Manhattan) at 36 agents.

The inconsistency in figure 6 for 30 and 36 agents is due to an issue in using an average fps. As agents are removed due to them reaching their goal, the fps increases because the code needed to run that agent is no longer needed to be ran. This causes the time needed to complete all tasks for a given update to decrease. At the lower numbers of agents, there would not be as much of an effect. This is because they will all be quite close to each other and therefore there is not enough time for the average fps to be affected by the lowering the number of agents. However, as the number of

agents gets larger, the distance between the “first” and “last” agent to finish gets longer. If the distance is longer then it means any change in fps will have a greater effect on the average because it will have an effect for longer. This means that if the first agent finishes there is a small increase in fps and the more agents there are, the more increases there can be, so for every agent that gets destroyed, the fps increases. The longer the distance the more these increases have an effect, to the point you have the anomalies shown by figure 6. This could have had a knock-on effect to the time taken but without investigating in much more detail it is unknown whether this is the case.

The only way of limiting the affect would be to stop the average fps as soon as the first agent finishes, but this would not be an average of the entire time, it would only be an average for an amount of time. However, the decision was made to not do it this way because the average fps should go for the duration from start to end, not just for a specific amount of time.

Overall, the results clearly show that the algorithm one runs at a much higher fps in all cases and that algorithm two is not as efficient as algorithm one.

Figure 6 also shows that for algorithm 1 as the number of agents increase, the lower the average fps. However, for algorithm 2 there is the same trend except for the 30 and 36 agent results. The prediction was that using a Manhattan heuristic would mean that the fps would be higher no matter the algorithm because the algorithm has no bearing on how many nodes are explored at a point in time. For algorithm 1 Manhattan is better at all number of agents. This was to be expected because Manhattan does not need to explore as many nodes, at once, as Euclidean does. Euclidean is very intensive in comparison to Manhattan, because Euclidean can explore in eight directions while Manhattan can only explore four. However, the inconsistency is the same inconsistency mentioned in the previous section. Due to that fact, it will not be explained again.

While algorithm 1 and 2 show a similar trend, algorithm 2 starts at a much lower average fps than algorithm 1. Algorithm 1 starts in the late 300s, roughly 375, while algorithm 2 starts at under half that at about 150. The same can be said for at 36 agents where algorithm 2 shows an end average of about 30 for both algorithms, whereas algorithm 1 shows an end average of 50 for Euclidean and about 75 for Manhattan.

Overall, the results show that Manhattan runs at a higher fps in most cases than Euclidean and that as the number of agents increases, the closer the two heuristics get on average.

4.2.3 Conclusion of the Evaluation

The results help form a basis for making an educated guess based on the data which is the best combination for a games developer. The best combination for a game’s developer is a combination that has a high number of frames while not taking long to complete. The reason this is important is because players will not put up with a low number of frames per second and not taking long to complete means that a developer can tune the difficulty of the agents or AI easier than if it starts at a high amount. Therefore, the best heuristic based on the results is a Manhattan heuristic, due to consistently being at a higher fps than Euclidean. The best algorithm from the ones compared in the results is harder to come to a definitive answer too, however in terms of time taken to complete algorithm 2 is best. On the other hand, in terms of fps algorithm 1 is better. Therefore, the best combination is the Manhattan heuristic with algorithm 1 due to the high number of frames it runs at while being the second fastest in terms of time to complete. Algorithm 2 with a Manhattan heuristic would have been the best if it did not have such a low fps.

The graphs shown in figures 5 and 6 were thought to be of the upmost importance as this is an easy way to know what is trying to be shown. The graphs were made with all related information on one

graph so that it was all together and can be all compared to each other. The lines were then colour coordinated across the graphs to keep them the same and to stop confusion between graphs.

The overall experiments were performed well, with each result being done three times. This allowed for an average to be taken meaning that the results would be more accurate to the true representation. However, anomalous results should have been removed from the average because they had a larger impact on the results than expected. On the other hand, there was not a way of knowing which results were the anomalous ones without doing the experiments many more times each. This would have taken more time that was not available. However, if there were more time then at the top of the list would have been to do the experiments more times and take out any anomalous results.

An issue that kept appearing was Unity and the computer that the experiments were done on. As the experiments went on, it was clear that the first run of a new number of agents, would be significantly slower than any run after that point. For example, there would be a change from 20 to 30 agents, the first test would be slower than the second or third. This meant a redo of all tests done to that point, which was about half, where the first test was discounted due to its significant difference. The reason that Unity and the computer are cited for causing this is because it happened in almost all cases, so the code itself had no bearing on whether this issue occurred. This issue could have also caused the anomalous results in the previous paragraph but there is no evidence to suggest that this is the case.

Another point to take note of is that there are ways of making the results more efficient through coding means. This was a first attempt at anything like this, so many mistakes were made meaning the results are not going to be as accurate as would have been preferable. This is something that would get better if there were a chance to do this again. For example, removing anything that was not relevant to the part that was tested. This was done in some ways, like removing the decomposition algorithm while testing was being completed. However, any global variables that related directly to the algorithm were not removed which would have taken up very little but still some processing power.

4.3 Evaluation of the Plan

4.3.1 The Original Plan

Below shows the original plan used in the project proposal. The green areas were used to show implementation. The yellow areas were used to show research. The red areas were to distinguish where testing was taken place. The inclusion of written pieces in the plan was thought to be important and this is shown by blue areas. The purple areas were used as a buffer in case of any unforeseen circumstances, which were especially prevalent with issues such as Covid being a constant potential problem.

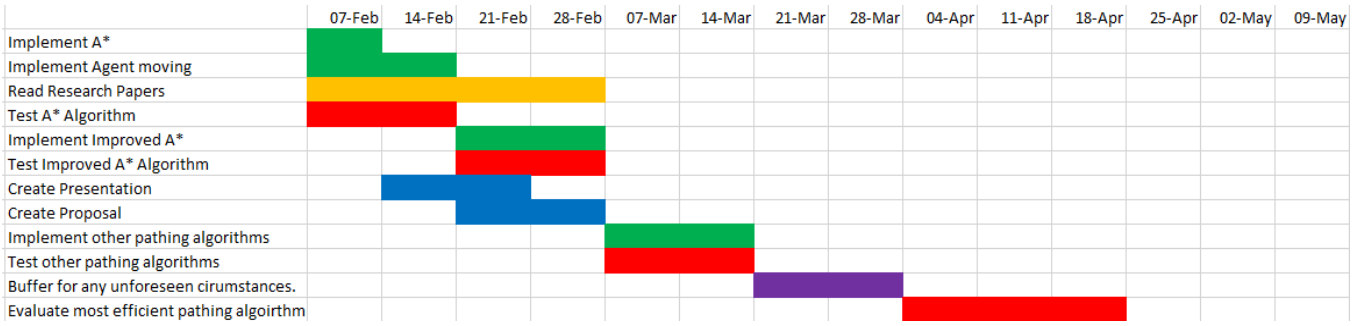

Figure 7 – Original Plan of Project.

4.3.2 The Inevitable Changes to the Plan

Plans are always made to be changed and this case was no exception. In the project proposal it was specified that it was known plans would change and that a more agile development process would be required. This is because there would be a higher chance of a successful project due to how things could change during the project. The implementation of A* and the agent moving was completed before this plan was created so this part of the plan was correct. However, after the implementation of A* and agent moving, the plan had to change as the project continued. The original plan was to focus on an improved A* during the "implement improved A*" stage however, it was discovered upon further in-depth research and implementation that this approach was not created for a maze but for an A to B situation. The only changes for the line occurred where an obstacle was in the way, in which it would move around the obstacle using obstacle avoidance. Therefore, this section was scrapped and replaced with changing heuristics which filled the time that was previously used for the "implement improved A*" phase, however, this meant everything moved across one week due to the mistake.

The "implement other pathing algorithms" section was then planned to be used on two improvements that were discovered during research into the topic. However, these were very complex and intricate sets of algorithms and took much longer to implement than planned for at the time. It meant that only the decomposition algorithm was implemented, but the algorithms that use the decomposition algorithm were not completed.

There was also a requirement for UI details to be added so that when the project was running, someone could understand some things that were happening such as showing the frames per second and showing how many agents had completed the maze. This took up a very small portion of time, but it was not included in the plan.

4.3.3 Sticking to Parts of the Plan

There were some parts of the plan that were stuck to, whether that be accidentally or on purpose. The part of the plan that was stuck to accidentally was the buffer. The buffer was placed starting the week of the 21st of March and ended at the end of the week starting the 28th of March. This buffer was a just in case something happened and as it got to the 21st apart from one week that became unproductive, nothing had substantially happened. This meant I was slightly ahead of where I wanted to be once changes to the plan had been made. However, on the week starting the 28th of March Covid made the next two weeks impossible for work to get done. The buffer was there for that reason as well as many others and it was good that it was since this happened.

The part of the plan that was stuck to on purpose was the writing parts of the project. It was known how important these were and especially how long writing can take. The dissertation itself was going to take quite a long time to write but one requirement was that for certain parts, the algorithm work had to have been completed. However, others did not need to be completed to start, so those were prioritised at the start while the algorithms were being finished. Keeping to the plan about writing the dissertation was tough because of the affect issues had on the plan throughout the project but the plan was kept to for that part, and it made the writing much easier and less stressful.

4.3.4 Evaluation Conclusion

Overall, it was known the plan was going to change. It changed a lot more than was expected but parts of the plan stayed the same throughout. This was a good lesson that changing a plan was not a bad thing and that it was inevitably going to happen, but some parts needed to stay the same to make the situation easier to handle. A major success from a certain point of view was that time was

allowed for unforeseen circumstances, which meant that there was less amount of stress on whether the project was to be completed on time. While there were in total three weeks, where there were mistakes that made the time not as productive as it should have been, which could have been planned for and losing two weeks due to having Covid, the original plan had accounted for the possibility which shows forward thinking and some good planning. However, the planning could have been much better in many places and issues could have been avoided if the planning was more thorough.

4.4 Evaluation of the Approach

4.4.1 What Went Well

The start of the project began quite well. The approach at the start was to try an understand what different ways pathfinding algorithms try to achieve their goals. This started by researching the most known ones such as Dijkstra and A* on non-professional websites. The approach gave a good understanding before attempting to read higher level literature which would be confusing to someone who did not have at least a basic understanding. This approach led the project to attempting different heuristics, these being Manhattan and Euclidean. Once the literature on pathfinding was looked at, this meant that an attempt could be made, once a basic A* had been developed, to try one of these new algorithms from the literature.

Therefore, the belief is that this part of the approach went well due to how much information was gathered and how the understanding of pathfinding algorithms changed and improved.

4.4.2 What Could Have Been Improved

A constant issue that also needs to be addressed is the technology used. Unity was the best tool to show how it all worked due to Unity being a very visual tool as well as being accessible. However, Unity runs into performance issues that aren't always clear and this can be frustrating to solve. The inbuilt statistics during play mode gives some detail on what is going on with the frames per second and the CPU, but not what is causing it. This is where the profiler comes in. The issue is the profiler is hard to find if it is not known about and once there, it is very complicated to use effectively. However, this could be solved using Free and open-source software (FOSS) to effectively use the profiler. This approach was not taken during the project because it was unknown until writing the dissertation. Therefore, this could be used in the future to effectively use the profiler.

5 Conclusions

This section will go into comparing the aims and objectives with what was complete, what has been learnt over the course of the dissertation project and the future work that could be attempted by others or if there were more time to continue this dissertation project.

5.1 Comparison to Aims and Objectives

The aim of the project stayed the same throughout, this was because it was believed to be important and what was interesting. The aim of the project as stated in the first section was to:

- Analyse the performances of different pathing algorithms to determine which is best suited to a group of agents or AI in the gaming industry.

The objectives on the other hand changed as the project progressed due to not fully realising how much work had to be done in the time allotted for it.

5.1.1 Objective 1

Objective 1 was not changed due to it being a core part of the analysis. Objective 1 was to identify two aspects of pathing algorithms which can be compared to determine a ranking of A* along with other pathing algorithms. These became time taken to complete the maze and average fps during the completion of the maze. There were no problems with this because these are two key components to the video games industry. The skill developed during this objective was critical thinking because a decision was made by analysing observations made from experience and the fact that people who play video games want a game that runs at a high fps. The two aspects discussed formed the results and is shown by Figures 5 and 6 where the aspects are the x axis. There could be further work to be done here by analysing an entirely different aspect of pathfinding in video games such as whether the algorithms truly get the shortest path.

5.1.2 Objective 2

Objective 2 was to design and implement an A* algorithm and test to make sure it performs path-finding tasks adequately. This did not change because testing was done to make sure that the path-finding algorithm was identifying a path that was heading from the start, where the agent was, to the end goal. There were some problems at the start of the project. For example, the video used to help implement the algorithm had errors in it, which meant that the skill of problem solving was improved to try and fix the issues which were not fixed in the video. Also, there was a misunderstanding in implementation which was swiftly corrected and then labelled as a new algorithm. The results achieved were the two different algorithms that could be compared, and these are shown in figures 5 and 6. The only future work for this section would be to remake the A* to be more efficient and to refine the efficiency as much as possible.

5.1.3 Objective 3

Objective 3 was to analyse the performances of the A* algorithm using different heuristics which did not change as seen by the results shown in figure 6. There were not any problems with using the different heuristics, however, there were some issues getting results. Since the results were measuring a form of efficiency, extenuating circumstances had to be limited. One such extenuating circumstance was that the first test of each section would be significantly slower than every test after that point. This was not picked up as an issue at first, but once realised, all results had to be retaken, therefore the second, third and fourth result were used instead of the first three. This eliminated an anomaly that could have skewed the results. The main lesson to be learnt from this objective is the analysis of the results. Analysing is a very important skill that can be used in a lot of different situations. The analysis can be seen in 4.2 Evaluation of the results where the results show that Manhattan is better for the average fps.

The results were taken to a standard that is believed to be the best that were possible. Using three results and then taking an average of these results is a very good approach to take to getting results because it gives a more accurate and viable result. This led into making several graphs, each increasing in usefulness to the point of having figures 5 and 6. These graphs took some tweaking to get right but the approach to it went well. Tweaks were made to increase visibility between the lines, decreasing the amount the intervals went up by on the x axis so that by a glance a rough estimate could be made to see the difference between two lines in terms of a value.

This project could be expanded by using other heuristics than the two most common. One example of this is the Hamming distance, which “is one of several string metrics” (i2tutorials, 2019). The

Hamming distance is defined as calculating the number of difference in positions between two equal-length strings of symbols (Nilsson, 2010) . Another example which builds upon Hamming distance is the Levenshtein distance, another string metric. The string metrics could also be investigated to see if they could be more efficient than the heuristics investigated in this dissertation. This could be done by comparing the most direct path with a path created, to see if there is a way of changing segments of the less efficient path to make it as efficient as possible to the direct path without breaking the rules set, such as walls. Whether this is possible is unknown, but it would be an interesting avenue to investigate.

5.1.4 Objective 4

Objective 4 was partly fulfilled since an algorithm, which was an improvement, was implemented. However, it was not from a research paper but was from obtaining a better understanding of how A* is implemented in today's standards. Therefore, objective 4 was not completed because trying to implement state-of-the-art algorithms was a very difficult process. This has improved technical skills because the algorithm was of a much higher level than had been done up to this point.

Trying to implement state-of-the-art algorithms has many issues that need to be overcame. For example, there are no tutorials online in video or text format. There is also no one asking about it on forums, so it is hard to get an idea of how to begin or how to solve issues if they occur. Due to the algorithms being in published papers, it was a relief that there was an answer, it was just about finding a way of getting to it. While the algorithms were not completed, an attempt was made, and the decomposition algorithm was successfully implemented which gave a sense of accomplishment.

Another reason why objective 4 was not completed was because of reading, understanding, and translating pseudocode. Pseudocode always was an issue, due to how complex some pseudocode can be or to how much it can leave to the developer to infer. The belief before this project was that pseudocode was a simpler way of explaining an algorithm that was somewhat like how a manual would. However, now it has become much more of a rough guide, it is only some pieces of the puzzle, not a manual on how to put the pieces together. For example, the pseudocode used to implement algorithms in this project did not define variable types or explain how to implement certain parts of the algorithm that would be required by C#.

Objective 4 could most definitely be considered as future work since it was incomplete. Finishing these algorithms could prove to have interesting results. If then connected to objective 5 this could form interesting results.

5.1.5 Objective 5

Objective 5 is like objective 4 since they directly lead on from one another. Objective 5 is partially fulfilled because it did analyse the performance of an improvement using the same heuristics as in objective 3. However, it was not from a research paper like first intended. Therefore, objective 5 was incomplete. The reason objective 5 was incomplete came down to similar reasons as stated for objective 4 and due to the time constraints and difficulty.

Objective 5 could have been successful if part of the approach had been improved. This improvement was implementing algorithms that were more well known. AAAI is a well-known and respected conference meaning it will have state-of-the-art algorithms that will be complex. The AAAI algorithms were very complex due to this being the first time working on something of this high

level. This also meant that the pseudocode was not simple or easy to understand at first. Due to my current experience, not many pseudocodes have been implemented meaning it was difficult to try to implement a high-level pseudocode. Therefore, working on a lower-level algorithm could have meant that it was completed and could have been used as comparison in the results section. However, another improvement to keep the algorithms could have been to have had practice with pseudocodes before and to have had experience at this high level. This was not the case during this project, but hopefully before future projects, practice can be done to improve those skills more and the skills can translate to future high-level projects. This practice

What was learnt is to try and recognise how much work can get done in a specified amount of time. This is something that will take a lot of work to master, but this project was a good start on that journey. Objective 5 could also be considered as future work since it was incomplete. Comparing the algorithms to the results shown in figures 5 and 6 could prove to have interesting results and comparisons.

5.1.6 Objective 6

Objective 6 was an objective made to further the work if there was time to do so. However, due to time constraints and issues throughout the project, objective 6 was not completed at all. This was believed to be a good objective to have since it showed further planning if the work was completed quicker than expected. However, the lessons to be learnt from this is that even though further planning is a good skill to have, knowing how long the project will take is better to know. In hindsight, this objective would be removed due to being incomplete and how long the project took to get to the point it is at.

However, due to the objective being about future work, if different algorithms were used and tested against the results, then this could prove for a very interesting comparison. If there were an algorithm out there that was undiscovered during this project that is an improvement, then it should be shown off and compared to the standard used by the gaming industry and could give developers are much larger range to work with in case the one they would normally use is not effective to their requirements. This includes the AAAI algorithm because it could make a large difference to many games when it comes to pathfinding. The AAAI algorithm is a state of the art, very well thought through algorithm that exhibits potential to be used in a plethora of games. A game such as Orcs Must Die would be excluded from this due to the nature of how the map changes all the time. However, other games with more static maps, at least where an entrance or exit would not be locked off at runtime, could really benefit from the two algorithms in the paper.

Therefore, overall, the project could be seen as partially successful due to not all targets being hit. However, this is something to learn from and can be improved upon. The lesson is that it is okay to not hit all the objectives but if you know why it happened and can see how it is possible to navigate these issues better, it was a successful and important use of time.

5.2 What Was Learnt

This project has improved a lot of skills both in terms of technical, computer-based skills but also in terms of soft skills.

In terms of soft skills, there are many skills that have been improved upon over the course of the project. One example of this would be problem-solving. This is a key skill for a programmer because that is most of what we do. Every error fixed, every implementation completed is problem-solving on various levels. Therefore, working on a state-of-the-art algorithm pushed problem-solving abilities a lot. This also links to critical thinking because thinking about an issue or implementation critically can make the problem solving easier. However, critical thinking was also improved when it came to research and analysing the data from the results obtained. Critical thinking was also improved because a decision had to be made on which was the best combination from the data that was obtained.

Communication is another soft skill that was improved because before this dissertation, there had been times where a write up was required but not anything like this dissertation piece. Therefore, learning how to appropriately communicate in a long form document was important. This involved reading other papers not just for their contents on pathfinding but also how they communicated the information they were attempting to communicate. Communication will be an important skill to keep improving, especially for when it comes to work environments. While the communication may not be in the same form, the skill of learning how to communicate will be instrumental in future endeavours.

5.3 Future Work

The future work suggested in this next section could also be combined with future works already mentioned. This would result in a highly detailed and interesting report on how different the outcomes could be.

5.3.1 Variety of Maps

There are many ways to take this work in the future. One path that could be taken is comparing the algorithms specified in this dissertation on a large variety of maps. The main map that has been used does not have more than one distinct solution, therefore this could be interesting to see what would happen if there were more than one distinct path. The results of this project show only one path and how the agents get to a single end goal, but if the agents had a choice, there could be a difference in the results that is unseen by the current implementation. This could also fit with making much larger maps with many more nodes. Having more nodes means many more different paths an agent could take but also more nodes for algorithms such as A* to traverse. This also means that algorithms are tested to their limits, which could prove to have varying results. The adverse effect of this is that the algorithms will take much longer to complete and will slow down the project due to the need to traverse many more nodes. However, as games are getting larger and larger in scale, this could be where pathfinding for the gaming industry is going in the future.

The improvement or opportunity that is expected by going down this future work is that the algorithms are proven to work on more than just one map and that different maps could produce different results. This could create a clear best combination or create best combination under specific criteria. Example criteria could be for a small group of agents over a large map.

There is also the idea of maps changing at runtime. There could be obstacles in the way that are placed at runtime, like how the Orcs Must Die series works, or walls could change position or rotation. This change would mean that algorithm 1 might struggle in this more dynamic scenario and may be unable to get to the end due to currently only stopping if the node the agent is trying to move to is “claimed” or an agent is on it. It may not currently account for obstacles or moving walls, so this would be an interesting way to go in further work. Algorithm 2 on the other hand would not

struggle since it runs the path finding for every attempt to move. This would however be harder to obtain results for as it would bring some randomness that would need to be removed by lots of results and taking an average of the results.

5.3.2 Increasing the Number of Agents

Furthermore, the number of agents used in this project is on a smaller scale than what some games require. Some games have up to hundreds of agents moving at once. Therefore, finding a point where the algorithm is unable to be used for a specific number of agents would be very interesting to developers. This is because they would know which algorithms are best for the parameters they are working within. For example, if they only need 10 agents moving at once then they don't need to be as strict with how many nodes are traversed within a certain period. However, if they need 100 agents moving at once then they need to be very strict with what how many nodes are traversed within a certain period and certain algorithms are better within those requirements than other algorithms. This would be an important improvement because it would focus on helping developers in a way that is meaningful.

No specific outcome of the experiments made this a potential future work, but the original idea for the project did. However, the expected outcome from increasing the number of agents is that a more comprehensible guide could be given and that developers have an example of how to use pathfinding effectively for larger groups of agents.

Bibliography

Bergami, G., Maggi, F., Montali, M. and Peñaloza, R. (2021) 'Hierarchical embedding for DAG reachability queries', *IDEAS '20: Proceedings of the 24th Symposium on International Database Engineering & Applications*. Online and Seoul, Republic of Korea. pp. 118-126 or 1-10. Available at: https://dl.acm.org/doi/10.1145/3410566.3410583.

Björnsson, Y. and Halldórsson, K. (2006) 'Improved Heuristics for Optimal Pathfinding on Game Maps', *Second Artificial Intelligence and Interactive Digital Entertainment Conference*. Marina Del Rey, California. pp. 9-14. Available at: https://www.aaai.org/Papers/AIIDE/2006/AIIDE06-006.pdf.

Cui, X. and Shi, H. (2010) 'A*-based Pathfinding in Modern Computer Games', *International Journal of Computer Science and Network Security*, 11, pp. 125-130.

Daniel (2018) *Unity - A Star Pathfinding Tutorial*. Available at: https://www.youtube.com/watch?v=AKKpPmxx07w (Accessed: February 7th).

Entertainment, R. (2012) *Orcs Must Die!* Available at: https://web.archive.org/web/20120504002825/http:/www.robotentertainment.com/games/orcsmustdie/ (Accessed: May 3rd).

Flam, A. (2018) *Pathfinding algorithms: Graphs*. Available at: https://shapescience.xyz/blog/pathfinding-algorithms-graphs/ (Accessed: May 7th).

Flick, J. (2020) *Frames Per Second Measuring Performance* Available at: https://catlikecoding.com/unity/tutorials/frames-per-second/#:~:text=We%20measure%20the%20frames%20per%20second%20each%20update,%7B%20FPS%20%3D%20%28int%29%20%281f%20%2F%20Time.deltaTime%29%3B%20%7D.

He, Z., Shi, M. and Li, C. (2016) 'Research and application of path-finding algorithm based on unity 3D', *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*. Okayama, Japan. IEEE. Available at: https://origin-ieeexplore.ieee.org/document/7550934.

Hu, J., Wan, W.g. and Yu, X. (2012) 'A pathfinding algorithm in real-time strategy game based on Unity3D', *2012 International Conference on Audio, Language and Image Processing*. Shanghai, China. IEEE. Available at: https://origin-ieeexplore.ieee.org/document/6376792.

i2tutorials (2019) *What is the difference between Euclidean, Manhattan and Hamming Distances?* Available at: https://www.i2tutorials.com/what-is-the-difference-between-euclidean-manhattan-and-hamming-distances/ (Accessed: May 7th).

Ide, J. (2020) *Orcs Must Die! 3 Review: Monster murdering mayhem is one of the best games on Google Stadia*. Available at: https://www.mirror.co.uk/tech/orcs-must-die-3-review-22356625 (Accessed: May 3rd).

Ju, C., Luo, Q. and Yan, X. (2020) 'Path Planning Using an Improved A-star Algorithm', *11th International Conference on Prognostics and System Health Management* Jinan, China. IEEE. Available at: https://ieeexplore.ieee.org/document/9296641.

Lexico (2022) *Definition of Heuristic*. Available at: https://www.lexico.com/definition/heuristic (Accessed: May 7th).

Metacritic (2011a) *Orcs Must Die! for PC Reviews*. Available at: https://www.metacritic.com/game/pc/orcs-must-die! (Accessed: May 3rd).

Metacritic (2011b) *Orcs Must Die! for Xbox 360 Reviews*. Available at: https://www.metacritic.com/game/xbox-360/orcs-must-die! (Accessed: May 3rd).

Metacritic (2021) *Orcs Must Die! 3 for PC Reviews*. Available at: https://www.metacritic.com/game/pc/orcs-must-die!-3 (Accessed: May 3rd).

Nilsson, N.J. (2010) *THE QUEST FOR ARTIFICIAL INTELLIGENCE*

A HISTORY OF IDEAS AND ACHIEVEMENTS. Cambridge University Press.

Pearl J (1984) *Heuristics: Intelligent search strategies for computer problem solving*.

TV, P. (2016) *C# Beginners Spice 02 - SortedSet and IComparer Interface*. Available at: https://www.youtube.com/watch?v=_UvTZk1HNbk (Accessed: May 13th).

Unity (2022) *Gizmos*. Available at: https://docs.unity3d.com/ScriptReference/Gizmos.html (Accessed: February 7th).
Waggener, B., Waggener, W.N. and Waggener, W.M. (1995) *Pulse Code Modulation Tehniques*. GoogleBooks: Google.
ZonaFree2Play (2011) *Orc Must Die, el juego de acción que no pasa de moda!* Available at: https://www.zonafree2play.com/2018/10/orc-must-die-el-juego-de-accion-que-no.html.