Division of Computing Science and Mathematics
Faculty of Natural Sciences
University of Stirling

Improving pathfinding in video games: An adaptation of the A* algorithm in a python-based video game

Ania Kottermair

Dissertation submitted in partial fulfillment for the degree of
Master of Science in Artificial Intelligence

September 2021

Abstract

Problem: Pathfinding is an important aspect of all video games. Algorithms like the A* are charged with mapping out the movement paths of non-playable characters and enemies. However, often these algorithms can come with many limitations such as slow speed, high CPU usage or the creation of suboptimal paths.

Objectives: The aim of this project was to improve the speed of the popular pathfinding algorithm A* in a python-based video game.

Methodology: The game used is a Pygame maze game, where the player has to create a maze through the placing of blocks that a spaceship has to travel through. All other improvements were made through adaptions to the Python code.

Achievements: The first improvement attempt involved adapting the weighted A*, which lead to minimal time improvements. The second improvement attempts adapted the way the A* algorithm chooses the f-cost from the open set, by either choosing one randomly or choosing the node with the highest f-cost. The latter proved to provide significant time improvements over all other algorithms. However, its limitations were discovered during a testing phase, showcasing that the improved algorithm creates suboptimal paths in open space maps.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this dissertation reports original work by me during my University project except for the following (adjust according to the circumstances):

- The Pygame video game by Alex Ter-Sarkisov taken from [1]
- The algorithm comparison tool by Xueqiao Xu taken from [2]

Signature: Ania Kottermair Date: 30/09/2021

Acknowledgements

I wish to thank Dr Andrew Hoyle for all the support and help he provided during this dissertation, as well as all my family and friends. I would also like to thank Alex Ter-Sarkisov for the Pygame video game he created which was a central part of this project.

Contents

List of Figures

1 Overview and Introduction

1.1 Introduction

The video game industry is one of the fastest growing sectors with recent years showing growth that even surpasses the film industry [3]. With this comes a desire for continuous improvement of the games, such as graphics, storylines and most importantly the game-player experience. Most players nowadays desire more realistic experiences, which are not only achieved through better graphics but also the best game artificial intelligence (AI), which ultimately will play into the success of the games [4].

In this dissertation thesis, we will investigate an important part of almost all video games, which is pathfinding. Pathfinding is the way for a computer to find the shortest path between two points, also known as nodes [5]. It is not only used in video games, but also in a large variety of fields such as robotics, GPS systems and more [6]. Most often in video games the pathfinding algorithms will be charged with finding the shortest route from a Mob (a hostile enemy character) or NPC (non-playable character) to the player or a second location. Given the importance of such a vital aspect of video games, optimising it could greatly benefit the industry in general. Pathfinding that is faster, more reliable and has lower computational usage will greatly enhance the user experience and overall increase player satisfaction.

Generally, pathfinding in video games specifically is lacking research with not many recent papers covering the optimisation of such algorithms [25]. In fact, it appears that most game developers will use the same handful of algorithms while only optimising them slightly for their use. The goal of this investigation will be to firstly analyse a variety of traditional pathfinding algorithms such as Dijkstra's Algorithm, the A* (read as A star), Breadth-first Search and Best-first search. Then, one of these will be picked to be optimised and improved and subsequently adapted to a python-based video game. Due to the nature of the programming language python, the game chosen will be rather simple in nature and be maze-based to truly put the pathfinding algorithm to the test. The game used will be an adaption of the Pygame space game by Alex Ter-Sarkisov [1].
The aim is to discover whether a well established pathfinding algorithm can be improved and adapted to a video game scenario. Pathfinding algorithms have a lot of potential for improvement, however such improvements are not well researched for video games. The objective of this thesis will be to analyse whether such improvements can be applied to video games as well.

1.2 Scope and Objectives

To be able to fulfil the aim of this project, a selection of algorithms needed to be compared against each other. The A* was ultimately the algorithm that was chosen as it

performed the best, but also since it is one of the most popular ones in many fields. Different mazes are then set up in a python-based video game which uses the A* algorithm to guide a spaceship through a maze. A set of metrics are then tested and used to improve the A*.

The simplistic nature of the game will make the research process much easier as it will use less computational resources. However, the idea is that findings could potentially be used in much more modern and complex games. The changes explored here could potentially improve the performance for even very complex games, possibly saving the player memory and CPU usage, as well as time.

1.3 Overview and presentation of the research question

The aim is to discover whether a well established pathfinding algorithm can be improved and adapted to a python video game setting. Research into pathfinding in video games is still very limited, with only a few recent papers covering the topic. Pathfinding algorithms have a lot of potential for improvement, with even very limited changes potentially having a great impact. The objective of this thesis will be to analyse whether improvements made to the A* can have an impact on the speed of the algorithm in a video game. Most video games, even very new titles, will still use classic pathfinding algorithms such as the A* for their pathfinding needs. The changes explored here could potentially improve the performance for very complex games, saving the player memory and central processing unit (CPU) usage, as well as time.

The overall question this thesis is therefore trying to answer is whether the pathfinding algorithm in a video game can be improved. Or more specifically, whether the performance of the A* algorithm can be improved in a python based video game and how these improvements could affect the performance of video games in the future.

This dissertation thesis will have a total of four chapters. The first chapter covered the introduction to the topic, as well as the objectives of the research conducted. The second chapter will cover the literary background and context such as a general background of artificial intelligence in video games and all important aspects of pathfinding in video games. The third chapter will then focus on the comparisons of the algorithms and the recording of the baseline values. The fourth chapter will cover the research process that was conducted in order to improve the A* algorithm in a maze-based video game. It will also cover the testing phase and results. Finally, the fifth chapter will include a discussion of the findings and a conclusion evaluating the work that was conducted as well as the impact it will have on future research.

1.4 Achievements

In this dissertation, knowledge of Python was required in order to run all the video games and any algorithm. Through in-depth research and familiarisation of pathfinding algorithms, the aim of the project was achieved and the A* algorithm was successfully improved. Through an additional testing phase the limitations of these algorithm improvements were also uncovered. This is especially useful, should someone choose to use any proposed algorithm in a video game or other project, as they can get an understanding of when it could work and when it would be best to look into a different algorithm.

2 Background and State-of-the-Art

2.1 Background on video games

The video game sector is continuously growing and improving [3]. The earliest video games such as Pong were more designed to be played with another player. However, as technological advancements progressed, the great potential of video games was uncovered more and more. Instead, of mirroring real in-person games that had to be played with a second person, video games had the potential of entertaining people on their own. Now, everyone could enjoy games alone when they had no one else to play with, fighting boredom but for some loneliness as well. With the exception of co-operative games, such as racing games or fighting games, all of which required an extra set of controllers that were often pricey, most games became solo experiences. The earliest examples were on the first computer systems, such as Spacewar for instance, gradually moving to dedicated gaming systems such as arcades and consoles. Some of the most notable examples include Space Invaders, Frogger and Pinball. Ultimately, with the internet merging with the video game industry, the focus was back on making video games multiplayer again, in some cases bringing together hundreds to thousands of players in the same game. However, both single-player or multiplayer games, as technology improved and became more complex, became more demanding. Game engines improved with non-playable characters needing to become more human-like and challenging to the player. Most games, especially when single player, have as the main focus the player's antagonists. In story-based or shooter games these are most often violent opponents who will try to eliminate the player, in racing games these will be other race car drivers that the player has to beat or in survival games these will be animals or creatures that have the goal to kill the player. These characters' behaviour in video games is most often generated through game AI.

2.2 Artificial Intelligence in video games

Generally, the main purpose of having AI in video games is to provide players with the most realistic experience, particularly in single player games [20]. The realism that a game provides will lead to greater immersion into the game, which in return results in the game being more entertaining to play. The greater the immersion and entertainment of the player, the more likely they are to keep playing it. For this reason, realism is of great importance to game developers who wish for the player to retain interest in their game. Also, having the game adapt to the player will allow more flexibility and freedom for them. This then allows for a more open game too.

Game AI can generally be understood as a broad set of algorithms or coding structures that will impact computer graphics, NPC or map behaviour and more. However, even though over the years video games have reached very high degrees of realism, game AI has not been following this trend. Generally, most game AI will be based on non-adaptive

methods, which means that there will not usually be any learning involved [10]. For this reason, game AI cannot be understood as AI in the traditional sense. The methods not adapting to user input will result in players often being able to exploit certain game mechanics to their advantage. For instance, hostile NPC behaviour will be manipulated as their patterns will become quite predictable to the player. Game AI would be charged with picking attacks or attack patterns which the player will eventually be able to recognise and outsmart. Similarly, a well-known example is that of the movement of enemies in the game Dark Souls, where players will abuse the movement algorithms of the AI to gain more experience points (XP). The player will get the enemy to chase them and subsequently lead them to a cliff where the enemy will fall down from and perish, while the player gains XP from the killing. This is also an example of how game AI is exhibited in a modern video game.

Game AI is not only used to mimic enemy behaviour but also ally behaviour. This will mostly be the case in first-person games such as "Half-Life", "Call of Duty" or "Overwatch". Where the role of an ally NPC cannot be filled by another human, game developers will design game AI that will mimic the behaviour of another player. Here again, it is of great importance to develop an AI that is realistic rather than perfect. This implies certain movements that a human would make, or for example shooting not always hitting the target or not perfectly dodging damage that is thrown their way. This helps the player assume human intelligence where there is none. This also ties into the fact that the game needs to be entertaining for players to keep enjoying it. If the AI ally makes the game too easy for the player, it will not be as enjoyable and immersive. There are many other examples of game AI such as changing the environment based on the players in-game decisions and even creating or changing dialogues. Yet, there will always be instances where the AI's behaviour will not match the game environment and appear inconsistent. This phenomenon is called "Artificial Stupidity" and will include behaviour patterns that humans might describe as lacking common sense or simply unrealistic. One example being in the game Pokemon, a single player game where the goal is to collect creatures that the player finds in the wild and fight the creatures of other "pokemon trainers". It is commonly known that often the AI that constitutes the pokemon trainers will pick attacks that make no sense in the fight that they are in. For example, they will pick a fire attack against a fire pokemon, instead of picking a water attack that might have more impact. For this reason, the game AI in the older Pokemon games is commonly described as being "artificially stupid" [11].

However, over the past few years with the increased desire to make video games as close to reality as possible, came also a desire to create the best human-like behaviour possible. Therefore, adaptive game AI has emerged with the aim of creating game AI that mimics the human behaviour of always being able to adapt to changes in our environments. This type of AI will use machine-learning techniques that enable the NPCs and the game itself to learn from the player's behavioural patterns.

One example, being the method of using dynamic scripting, as described by Spronck, Ponsen, Sprinkhuizen-Kuyper and Postma in 2006 [28]. This method enables hostile ene-

mies to adapt to changing player tactics through pre-defined elements. It will learn player behaviour and tactics when they are acted out repeatedly through an algorithm. For instance, it will scale the game difficulty to the player skill in a role-playing game. Other examples will utilise methods based on reinforcement learning and decision-making systems [24].

Nevertheless, adaptive AI is still not very practical due to the large amount of memory such a process would require and the amount of training that would need to be done for the AI to appear very realistically. The game would need to be quite long and repetitive in order to gather enough training data from a single player. A solution to this would be to combine the data from all players and share the trained knowledge. However, such methods are not very well researched yet. Thus, AI methods can be employed in many different ways, mostly using non-adaptive techniques as adaptive methods are not optimised yet. Another important AI method in video games that is often overlooked are pathfinding algorithms, arguably the most important non-adaptive game AI.

2.3 The concept of pathfinding

Pathfinding algorithms, while at first glance not seeming like a traditional AI technique, are an important aspect of many modern technological fields. Even though, they do not have the property of learning as do other AI techniques, they are still considered to have intelligent properties [27]. For instance, these algorithms give a non-sentient agent (should that be a NPC in a game or a robot in real life) the ability to navigate a space intelligently and naturally, avoiding both fixed and dynamic obstacles without requiring too much computational effort. This ties into the fact that most pathfinding algorithms have been developed by AI researchers.
Such algorithms can generally be found in a lot of fields. Most consumers will have encountered pathfinding in everyday life in GPS technologies such as in Satellite Navigation Systems or Google Maps. Such algorithms are also used in the field of robotics to help robots navigate 3D spaces.

2.3.1 General use of pathfinding algorithms

Generally speaking, pathfinding can be understood as the plotting of a path by a computer system between a start and an end node. This algorithm will repeatedly search the space until it finds the shortest path. However, most often finding the truly shortest path can often take a long time depending on the size of the space it is run in. Therefore, compromises had to be made in order to find paths faster. If a pathfinding algorithm finds a path much faster, generally this path will not be the shortest and most optimal. It will use techniques to smartly deduce which node it should travel to in order to find a path in a speedy manner that is still adequate. In a best-case scenario the algorithm sometimes even finds the optimal shortest path. However, no fast algorithm can guarantee

this with full certainty as this would imply that the algorithm knows every node and can confidently say that the path chosen is the shortest one.

The average person will have most likely encountered a pathfinding algorithm in their life. With the rise of mobile GPS devices, more users are relying everyday on Personalized Route Recommendations (commonly known as PRR) in dedicated devices such as Tom-Tom or Garmin Sat-Nav systems or mobile apps such as Google Maps and Waze [18]. Using online and satellite maps of road networks, PRR aims to generate for each user a path from their location to a chosen end destination [15]. Such pathfinding can be very challenging due to the many factors that need to be taken into account, such as environmental factors, personal preferences, temporal constraints and more. Such navigation systems will often exend upon existing popular algorithms, such as Dijkstra's and the A* algorithm to effectively analyse the search space it is in and produce a high-quality path for the end-user [18].

Another important application of pathfinding is in the field of robotics. Pathfinding in this field is conducted in continuous and unknown 2D/3D environments [14]. Generally pathfinding will start by creating a simplified grid-like view of the search space through skeletionization techniques [6]. Then, similar to navigation systems, it will use a pathfinding algorithm to find the end point from a starting location, the most common one used being the A*. What makes pathfinding for robotics so important is the real-life aspect it has. Similar to satnavs it will use the real world as a search space, however the agent travelling the created path is not sentient. Humans are most often able to avoid objects that might obstruct them in their path, however robots do not have this ability on their own. Therefore, the path created must take into account objects not only to create the path to the goal but also to avoid any damaging collisions. This is also quite different from pathfinding in video games where again the agent is not sentient and needs to avoid collision at all cost, however no real life damages can be done to themselves or the environment. Other applications include logistics and crowd simulation, where pathfinding algorithms will be used to control agents in real-life simulations to analyse concepts such as collision avoidance and multi-agent pathfinding [26].

2.3.2 Use of pathfinding in video games

As previously discussed, pathfinding is an important AI method in video games. It enables non-playable characters to appear more realistic and human-like. Similar to agents in robotics, the character is a non-sentient agent that needs to navigate an unfamiliar space while avoiding obstacles. However, in the case of video games, the search space is pre-defined and much smaller. It also does not have the truly unpredictable nature of human life, making the setup of the pathfinding algorithm much more straight forward. For instance, in a lot of older games obstacles will not move and other characters in the game can be passed through. However, as games become more realistic themselves and more visually demanding, pathfinding had to follow that trend as well. Some games can now generate random maps, making the search space as unpredictable as the real world.

Other games, will have a large amount of moving objects or characters that seem to appear in random patterns and locations, that the algorithm also needs to account for in some cases.

Depending on the game, the pathfinding AI can have different goals and characteristics. For a first-person shooter game, such as Bungie's "Halo", the algorithm would be in charge of leading the hostile enemy character to the player while avoiding any obstacles in the way. In a similar fashion, in role-playing games like Bethesda's "Skyrim", it would be friendly characters that could be led to the player. In stealth games, where the player has to sneakily avoid hostile mobs, the pathfinding algorithm would be charged with moving through the space in a predictable manner. The player is then able to analyse the enemy's movement while attempting to stealthy go passed it. An example of such a game is Naughty Dog's "The Last of Us", where the player is charged with avoiding zombies amidst an apocalypse. In race games, the algorithm is needed to control your racing opponents. For example in games like Nintendo's "Mario Kart", the enemy characters will not be pre-programmed to drive a certain path since they are also charged with bumping into and throwing items at the player, while the player drives in unpredictable ways. However, pathfinding is not only used to navigate non-playable characters through a space in a video game. In games like Rockstar's "Grand Theft Auto", the player has a map that can be used like a GPS to find a path to a certain location. This is especially helpful while driving a car, and the player is able to choose whatever location they wish to travel to. The map then forms a path from the current location to the chosen destination, which the player can then follow like they would on a real map.

2.4 Game world geometry

Typically, pathfinding algorithms can be presented on a simple graph or grid-like map structures. Grid maps will create a representation of the environment using small regular shapes called "tiles". Less complex games will generally go for such a representation of the world as it is simple and easy to understand for the pathfinding algorithm. An example of such a map representation can be seen here:

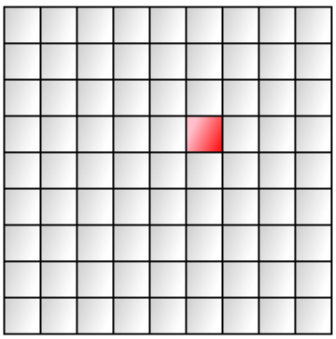

Figure 2.1: Grid Map

These types of maps will be used for more simplistic scenarios as they come with many implications. For one, the movement from one tile to another can vary greatly depending on the game. The agent could simply travel from one tile to another, always ending in the centre of the tile. Or it could also consider moving the agent on edges and vertices. Similarly, one may also wish to allow diagonal movement where the cost could be weighted differently. Also, if the tiles are particularly large when compared to the size of your agent, it could also be considered to move it from one edge to another. These considerations also need to be made when remembering that most of these maps will have obstacles that the agent will encounter. Often, these obstacles will end up also having edges and vertices that need to be considered by the algorithm depending on their size relative to the tile.

For map representations in more complex games the most common alternative is the use of polygons. This type of representation is typically used when the movement across large areas is uniform and if the agents can move in straight lines instead of going from tile to tile. Find below a small example of such a map:

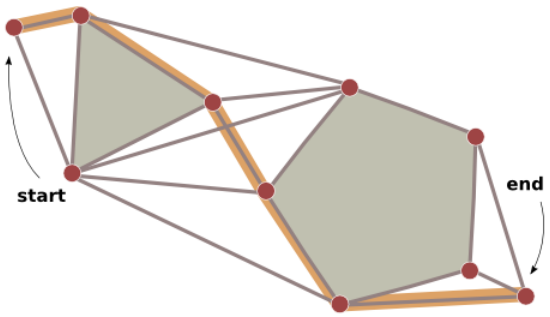

Figure 2.2: Polyglon Map

The example above shows a polygon map where an agent needs to travel from a start to an end point while avoiding two obstacles of different sizes. As can be seen, there are several waypoints, known as navigation points, that are defined on the map as seen on different parts of the polygons. For the pathfinding algorithm to know which points are connected, it needs to build a visibility graph. This graph represents pairs of points that can be seen from each other [8]. The purpose of this is to check the line of sight from existing vertices and add edges where needed. There are many simple algorithms that do this especially on more simplistic non-changing maps, however for more sophisticated maps a more powerful algorithm is needed.

As seen above, polygon maps are used in more simplistic game maps for pathfinding. However, more complex environments will most often use a similar concept known as navigation mesh (Nav Mesh for short). In such maps the obstacles are not represented as polygons, but the walkable areas are. The polygons in such a mesh need to be convex in order for the agent to move in a single line. Often in more complex games these

walkable areas can even include additional conditions such as height, or the requirement of swimming, or greater movement cost…etc. For navigation meshes, the agent can in the simplest form travel like on a grid where they travel from one centre to another, or through edges and vertices as navigation points. Find below an example of such a map:

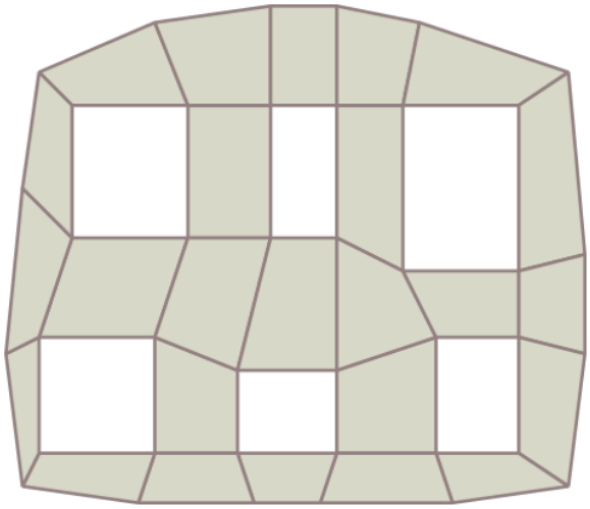

Figure 2.3: Navigation Mesh

As on a grid, the centre of the polygons can be defined as a first set of nodes in the environment. Edges and vertices can be used to define navigation points on the map. In addition to this, the algorithm will add the start and end nodes to the map that can be anywhere but the obstacles (as can be seen below).

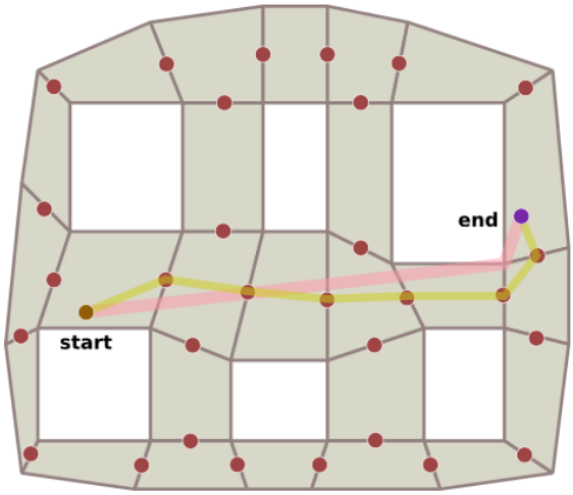

Figure 2.4: Navigation Mesh 2

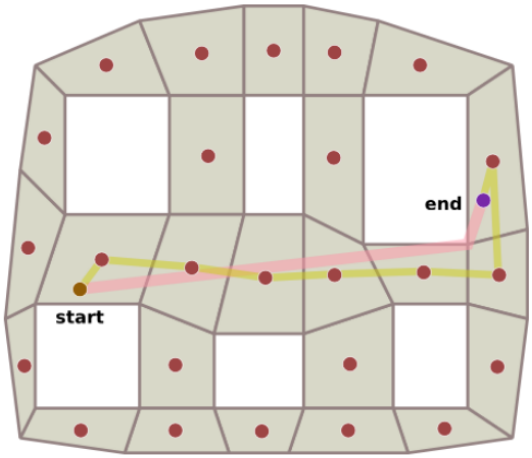

Figure 2.5: Navigation Mesh 3

2.5 Pathfinding algorithms

For an agent to be able to travel from A to B, an algorithm is needed that evaluates the space and defines the path. There are several algorithms that are commonly used, and some of the most popular ones will be discussed here.

2.5.1 Dijkstra's algorithm

Dijkstra's algorithm was the first response to the pathfinding problem and was the algorithm of choice for a long time [5]. It works by conducting an exhaustive search of the entire graph, meaning it searches the shortest path for every node from the start node. The complete process of the algorithm works as follows:

0. The starting node will be considered as solved.
1. Identification of all unsolved nodes that are connected to any solved node (in the case of the first step, the starting node).
2. For each line connecting a solved node to a unsolved node, calculate the cost.
3. Choose the node with the smallest cost. If there is a tie, then a random node is chosen.
4. Change the status of the node with the lowest cost from unsolved to solved.
5. Add a line to the set keeping track of the overall path.
6. Repeat steps 1–5 until the goal node is reached.

The cost for Dijkstra's algorithm is calculated as follows, where $g(n)$ is the cost thus far to reach node n:

$$f(n) = g(n)$$

This method ensures that the shortest path is always found. However, for larger maps or matrices, this can lead to great processing times and high computational efforts. In response to these issues, the A* algorithm was created.

2.5.2 A*

The A* algorithm is one of the most used algorithms for pathfinding. It can also be applied to many non-pathfinding problems in engineering and robotics for instance. It is very commonly used in video games as it performs very well and is easy to implement [12]. However, it can also suffer from high CPU and memory usage, as well as high processing times. These issues can ensue for a large variety of reasons, such as computational limitations as well as the heuristic function used.

In essence, the A* analyses repeatedly the most promising closest unexplored node until it reaches the end node. It compares well to Dijkstra's algorithm, as it is an improved version of it. Unlike Dijkstra's, which analyses all the nodes on the map, it will use an adapted cost function to determine whether a node is worth travelling to. This adapted function uses heuristics in its calculations. These heuristics add a layer to the previously discussed Dijkstra's function that enables the algorithm to run faster as it won't explore all the nodes. Once the cost for a step is calculated, the node with the lowest cost will then be picked. The process is then repeated until reaching the goal node. While doing so, the algorithm makes a note of all the nodes it encounters and adds them to what is known as the priority queue or the open set. For this reason, the A* has very large computational and memory requirements.

2.5.3 The cost function

The algorithm uses the cost function $f(n)$ that calculates the estimated cost of the path using the node n (i.e. the current node). The function looks as follows:

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

where

- $f(n)$ = total estimated cost of path through the node n
- $g(n)$ = cost thus far to reach the node n
- $h(n)$ = estimated cost from the node n to the end node.

Therefore, $f(n)$ merely produces an estimated cost for each node and is considered a heuristic function. The algorithm works by calculating the $f(n)$ cost estimate for all adjacent cells to the node where it is currently situated to then travel to the node with

the lowest cost. The process is repeated until the goal node has been reached. Now, to understand how the function works it is important to understand each component of the function. While $f(n)$ simply refers to the overall cost estimate, $g(n)$ represents the overall cost so far to reach the node in question n. Now, $h(n)$ can be understood and done in a variety of ways. The most common method used is the Manhattan distance. As mentioned previously the heuristic aspect of the A* is what makes it different from Dijkstra's algorithm. Thus, if in the A* $h(n) = 0$ it becomes Dijkstra's algorithm and will guaranteed find the shortest path.

Find below all the steps the A* algorithm takes when calculating a path.

1. Create the open set.
2. Mark the start node as open and calculate $f(n)$.
3. From the open set, select the node with the lowest f-score. Ties are solved arbitrarily, but in favour of the best node.
4. Otherwise, the current node with the lowest f-score is marked as closed. For all nodes adjacent to the current node, if it is walkable or if it is marked as closed, it is ignored. Otherwise, if the node is not currently on the open list, it is added and calculate the f-score for it. Alternatively, should the node already be on the open list, check if the path to that node is better using the g-score.
5. If the currently selected node is the goal node, it is marked as closed and the algorithm is terminated.
6. If the target square cannot be found and the open list is empty, no path is created and the algorithm is terminated.

2.5.4 The heuristics

Moreover, it is important to pick a good heuristic method in order for the A* to perform to the best of its abilities. In an ideal scenario, $h(n)$ would be equal to the exact cost it takes to reach the final node. This is evidently not possible as it cannot know the path beforehand. Another factor that is important when choosing a heuristic function is that it has to be admissible. This means that it can under no circumstances overestimate the cost to reach the end node. Examples of such admissible heuristics are Manhattan distance, Hamming distance, Euclidean distance and therefore also $h(n) = 0$. The Manhattan distance works very well with the A* as is calculates an $h(n)$ that is less than the cost of reaching the goal node. As can be seen below, it is computed by determining the overall number of squares travelled vertically and horizontally to reach the goal. This method will ignore diagonal movement as well as obstacles. It can be understood as follows:

$$h = |x_{start} - x_{destination}| + |y_{start} - y_{destination}|$$

Due to the nature of this method, the algorithm will prefer straight line movements. Another very popular heuristics method is that of the Euclidean distance. The function looks as follows:

$$h = \sqrt{(x_{start} - x_{destination})^2 + (y_{start} - y_{destination})^2}$$

Compared to the Manhattan distance it is far more precise and accurate, however due to this precision it is much slower.

2.5.5 Limitation of the A* algorithm

Nevertheless, the heuristics aspect of the A* is at the same time its greatest limitation. While it gives the algorithm a significant edge over Dijkstra's in terms of speed, it is also not as reliable if the goal is to find the shortest path. Since the heuristic is a mere estimation of the cost from the current node to the end point, it adds a level of uncertainty to the path. Since this aspect is an estimation, it renders the whole path an estimation as well. This means that f(n) overall becomes a heuristics function rather than a foolproof algorithm with an optimal solution. Yet, the Dijkstra's algorithm always examines the whole search space, leading to a stage where it will know all nodes on the map and therefore be able to deduce the truly optimal and shortest path to the goal. As can be seen in Figure (ref), the A* works with estimation from node to node, while Dijkstra's examines every node and then finds the shortest path.

However, in many cases the trade-off between speed and finding the truly shortest path can be argued as being needed. This is especially the case in video games, where the player would rather have the enemy come right at them even if this takes a couple more steps (something the player would probably not notice anyway), instead of having to wait for the game to compute the exact shortest path.

2.6 Other types of pathfinding algorithms

2.6.1 Breadth-first search Algorithm

The Breadth-first search algorithm is a traversing algorithm that visits each node of a graph or tree in a breadthward motion. The algorithm starts at a selected node and explores all the neighbour nodes before moving to the nodes at the next level. The steps are as follows:

1. Choose any node to be the starting node.
2. Explore and traverse any unvisited node that is adjacent to the starting node.
3. Mark any traversed node as completed.
4. Move to the next layer of adjacent unvisited nodes.
5. Repeat this process until all nodes are marked.
6. Create the path going backwards.

The Breadth-first search finds the shortest path from a given starting node to all other nodes. It has different applications, for example network analysis, map routing, as well as puzzle solving.

2.6.2 Best-first search algorithm

The Best-first search algorithm uses an evaluation function to choose the most promising node (from all available nodes) to traverse the graph to that node.
A variation of this algorithm is the Greedy Best-first Search algorithm, which first expands the node which has the smallest estimated heuristic cost to the goal node. It does however not take into account the cost thus far to reach the current node. The costs of the nodes is stored in a priority queue.
The A* algorithm is a Best-first search algorithm however it cannot be categorised as greedy as it takes into account $g(n)$, which makes it more optimal than the greedy Best-first search algorithm. The steps of the Best-first search a similar to those of the A*. However, what differs is how a node is chosen to be included in the path. It does not take into account the cost so far to reach the node and only traverses using the heuristic function.
Therefore,

$$f(n) = h(n)$$

This also implies that if the A* algorithm has $g(n) = 0$ then it turns into a Best-first search algorithm.

2.7 Improvements of pathfinding algorithms

Knowing how pathfinding algorithms work, it can now be understood how these could potentially be improved. Generally, the improvements of such an algorithm lead to either better processing time performance, better paths (meaning the shortest or most optimal path), the reliability in finding the best paths, lower CPU and memory usage, or all of the above. These improvements are generally made by either improving the hardware the algorithm runs on, improving the map, creating a new algorithm or improving the heuristics.

2.7.1 Improvement of the hardware

The simplest solution for improving the performance of a search algorithm is to improve the hardware the game and the algorithm runs on. A better CPU and more memory will result in the whole computational aspect running faster and having lower memory usage. However, in the realm of gaming this will be an aspect the user has to take care of. A player with a better PC for instance, will have a game that runs smoother and evidently a pathfinding algorithm in the background that runs at a much faster pace. Similarly for consoles, where younger consoles (e.g. PlayStation 3 vs PlayStation 4) will have better components which means the game will run much smoother. However, this is an expectation that cannot be met by every consumer meaning that the game developers will have to find a way of optimising their algorithms and ultimately their games.

2.7.2 Improvement of the map

One way of doing so is by improving the way the map is laid out or what type of map is used. A recent study conducted by Adaixo from 2014 used the influence map of a game, meaning the map that represents game information such as events or strategic and tactical decisions to optimise pathfinding in video games. This resulted in better time performance and less memory usage as well [7]. Similarly, a study from 2013 by Lee and Lawrence developed an entirely new pathfinding algorithm they called DBA*. It uses a database of pre-computed paths to effectively reduce the search time. This method claims to require less memory and time to compute, compared to real-time search algorithms such as the A* [13]. Another study by Cui and Shi from 2011 developed a method of direction oriented pathfinding, where the search space is laid out as an n-sided polygon navigation mesh, instead of a more conventional graph or grid, and popular algorithms such as the A* can then be used to find the shortest path [4]. This is supposed to propose a higher quality path and use less time and memory. Similarly, a study by Anguelov from 2011 also proposes a new algorithm bases on the A* called the Spatial Grid A*. This method uses grid-based navgraphs, a way of storing the entire game environment and spatial information. The algorithm stores this information as a bitmap image resulting in lower memory usage overall [8].

2.7.3 Improvement of the algorithm and heuristics

Generally, if an algorithms needs to be improved the easiest solution is to analyse the algorithm itself and how it is run. For any kind of algorithm, small changes to how it is coded or set up can lead to improvements in time as well as general performance. Over the years, there have been a few papers proposing improvements to pathfinding algorithms. For instance, a paper by Lina and Yungfeng from 2010 proposes an improved version of the A* that searches less nodes in order to ensure higher efficiency of the algorithm. This is achieved by introducing a new function that improves the heuristic aspect of the A* cost function [19]. Similarly, a paper by Mathew from 2015 proposes a similar method found on direction based heuristics that speeds up the overall search process of algorithms such as the A* and Breadth-first Search. It only evaluates only the necessary nodes on the map and eliminates redundant nodes from the graph, reducing memory usage effectively [5]. Another paper by Peng, Huan and Lao proposes a very unique way of improving the A* algorithm by improving the method of storing and accessing information in the open set. Their proposed improved algorithm is able to access information from the open set in one operation, as opposed to traversing the information of multiple nodes in the original A*. This proved to increase the operating efficiency by more than 40% [14]. There has also been a limited amount of research conducted into the development of adaptive algorithms for player and enemy movement. A recent study conducted by Wang, Wu, Zhao and Lin developed an empowered version of pathfinding algorithms that uses deep learning and machine learning to generate user-specific route suggestions [18]. However, such a method has not been adapted to video games just yet.

Finally, a common method that is often explored in regards to search algorithms is adding a weight parameter. This added weight to the heuristic score can be used to influence the balance between the quality of the solution and the search effort [21]. For the weighted A*, it is generally commonly assumed that increasing the weight on the heuristic estimate leads to faster searches overall. This is a parallel to the previously mentioned Greedy-best search that only uses the heuristic cost to create a path. This means that the higher the weight in the weighted A*, the more it becomes like a greedy search. However, this also comes with the added flaws of occasionally expanding more nodes than the A*, leading to paths that are not optimal. Also, it is known that increasing cannot always guarantee a faster search [22]. Moreover, it is important to note that not a lot of literature exists of the benefits of weighted search algorithms in gaming. For this reason, this project will investigate whether the weighted A* could increase the speed of the search.

Now, it is important to understand which improvements would benefit a video game the most. It seems from past studies that the most sought after quality of an improved search algorithm is speed and less computational usage, rather than path quality or optimality. These also make sense in video game context, as players would most likely prefer a game that runs faster, smoother and has lower CPU and memory usage, than have a game that always finds the shortest path in a pathfinding problem. Most games require focus and attention and generally immerse the player, leading to the average player not paying attention to smaller details such as the amount of steps it took for an enemy to approach them. And since these most players most likely value their PC resources and time over other factors, it can be assumed that these metrics are what need to be improved in a search algorithm.

3 Comparison of algorithms and recording of baseline results

This chapter will cover all the work that was done before the algorithms improvements and the testing phase. It will discuss the algorithm comparisons that were conducted until ultimately choosing the A*, as well as the recording of all the baseline values that will be used for comparison in the improvement and testing phase.

3.1 Comparison of algorithms

The research process began by deciding which algorithm should be chosen to be improved. To do this, some of the most popular pathfinding algorithms were chosen and compared against each other. The algorithms chosen were A*, Dijkstra's, Breadth-first search and Best-first search. Using the online tool by Xueqiao Xu, the pathfinding algorithms were put to the test on the same arbitrarily created map (see Figure 3.1) [2]. This particular algorithm tool was chosen to ensure all algorithms could be tested on the same map and in the same context for a fair comparison. The test map was designed to be large in size and with a significant amount of windy roads and dead-ends.

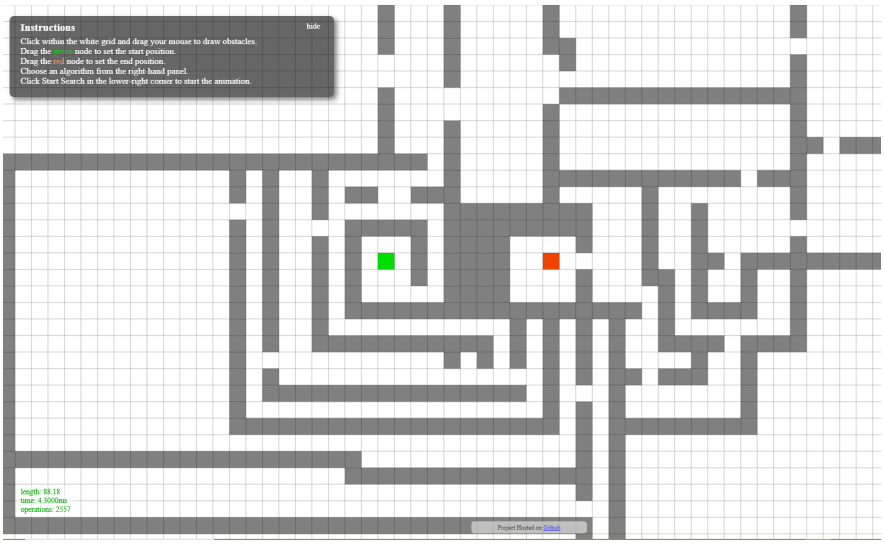

Figure 3.1: Algorithm test Map (green square = start node, red square = end node)

Several heuristic distances were tested as well for the A* and Best-first search. The results of this comparative process can be found in the table below:

Algorithm	Length (in cm)	Time (in seconds)	Heuristics
A*	88.18	7	Manhattan
A*	88.18	8	Euclidean
Best-first search	92.33	2.5	Manhattan
Best-first search	94.33	3.7	Euclidean
Breadth-first search	88.18	49	None
Dijkstra's	88.18	16.2	None

As can be seen above, the A* algorithm consistently outperforms the other algorithms by finding the most optimal shortest path in the shortest amount of time. It can be confirmed that this is the shortest path as it matches the result of the Dijkstra's algorithm which will always find the shortest path as it analyses every node. Nevertheless, the Best-first Search appears to be much faster than the A*, however it often will not find the shortest path.

Now, considering these results, it would be of the greatest interest to attempt to improve the A*. As it already outperforms the rest, it would be interesting to see if it can achieve similar times to the Best-first search, while keeping the quality of the path. Furthermore, as previously discussed, the A* is still one of the most used algorithms, not only in gaming, but in many other fields. Improving it could greatly benefit future games as well as potentially affecting other sectors too. Therefore, the A* will be chosen as a base algorithm for all incoming improvement attempts and tests.

3.2 Description of the Pygame-based maze game and the baseline mazes

As important as the choice of algorithm was the choice of which game the improvements should be tested on. Due to technical limitations, the game chosen had to be run in Python. The best platform of choice for games based in Python is Pygame. Pygame is a set of modules containing graphic and sound libraries used to create video games in the programming language Python. The game chosen was an open-source maze based game built by Alex Ter-Sarkisov. The game gives the player the ability to create a maze, which a spaceship then has to navigate. The player also gets to decide the starting position of the ship as well as the goal it has to travel to. A screenshot of the game is shown in Figure 3.2.

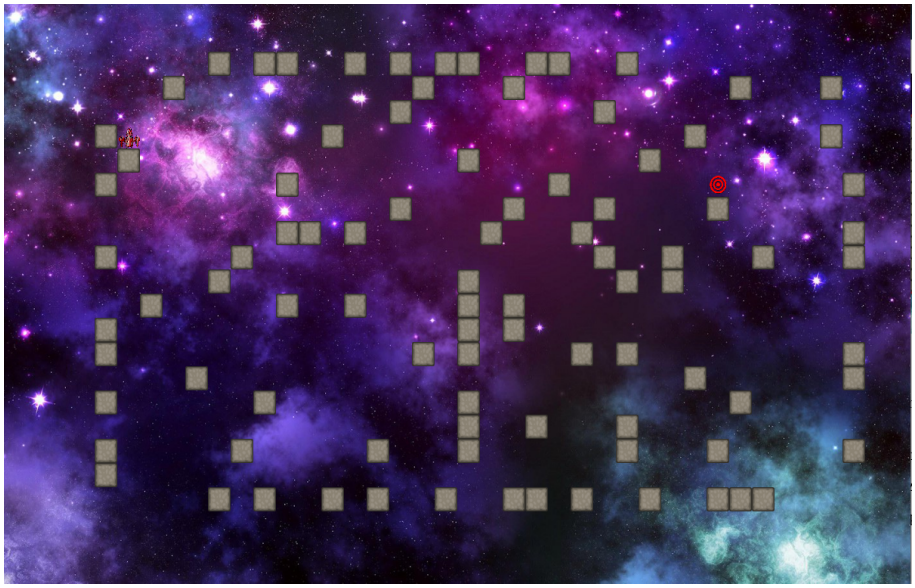

Figure 3.2: Example Screenshot of the Pygame-based video game by Alex Ter-Sarkisov

In order to ensure consistency across all tests performed, the background was altered in order to define a clear space where all mazes were built. The space was defined by a red box that was central to the overall background. This ensured the best computational efforts as the space was smaller, as well as easier construction of mazes. An image of this adjustment can be found in Figure 3.3.

Figure 3.3: Adjusted background of the game

For most mazes, the lines of this central space were used to define the walls of the mazes. Due to the game not containing a feature to save mazes, it was of great importance to

have a clear structure that could be followed to ensure all mazes could be replicated. For all improvements that will be made, each maze had to be run several times to gather time metrics that were then averaged and compared. These times would not be accurate if the mazes were not the same at every run.

3.3 The baseline mazes

As previously discussed, to increase player satisfaction the two metrics that would be of greatest interest are CPU usage and speed. This thesis paper will focus on improving the speed of the pathfinding algorithm. Speed in this case represents the time it takes for the algorithm to find a path from the start node to the goal node. Specifically for the game used, it represents the time after the maze is built and the player hits the "space" button on their keyboard until the algorithm finds a path on screen. Before any tests could be performed, base values had to be recorded for a total of five baseline mazes. These were used to record all the baseline values for the unchanged original A* algorithm, as well as the first values for the improved algorithms.

The first maze has a rather simple structure with long corridors and paths that lead to dead-ends that could confuse the agent. An image of this maze can be seen in Figure 3.4 along with the solution the A* finds in Figure 3.5.

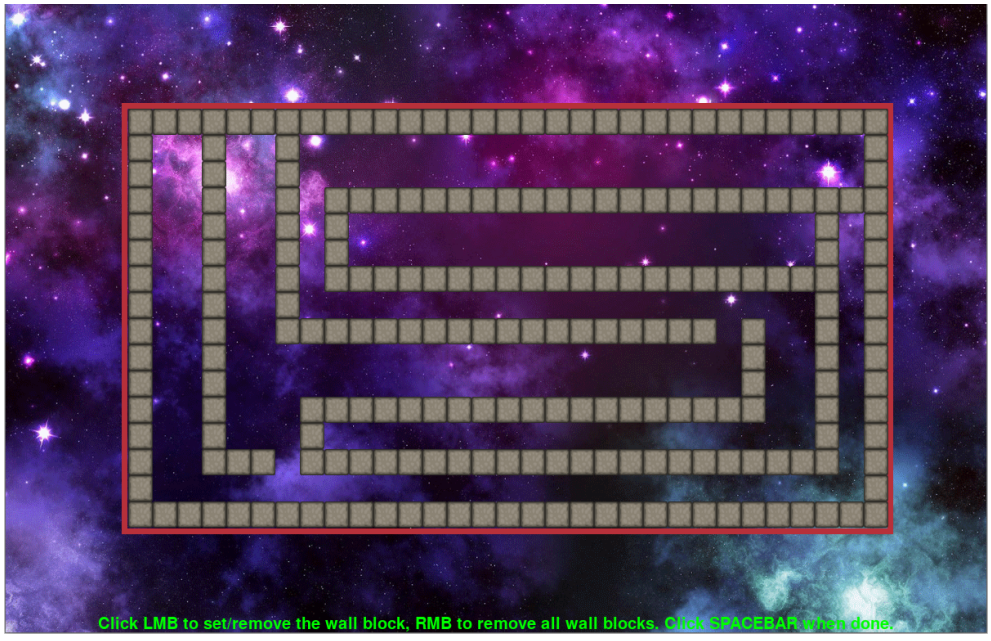

Figure 3.4: Maze no.1

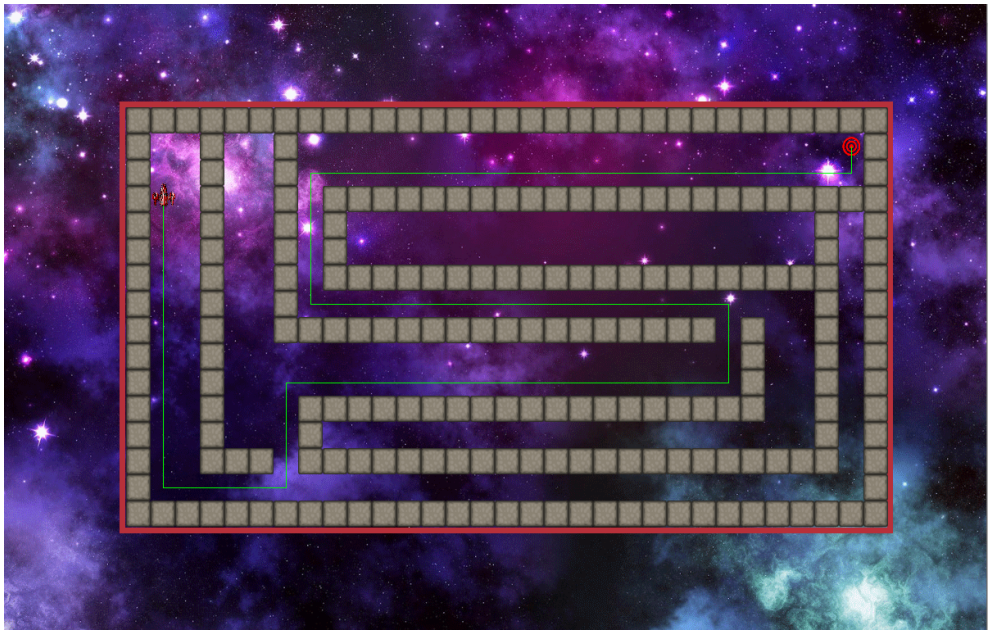

Figure 3.5: Maze no.1 solution

The second maze (see Figure 3.6) was built to have even more long and windy roads with several dead-ends and paths the agent could take. The solution can be found in Figure 3.7.

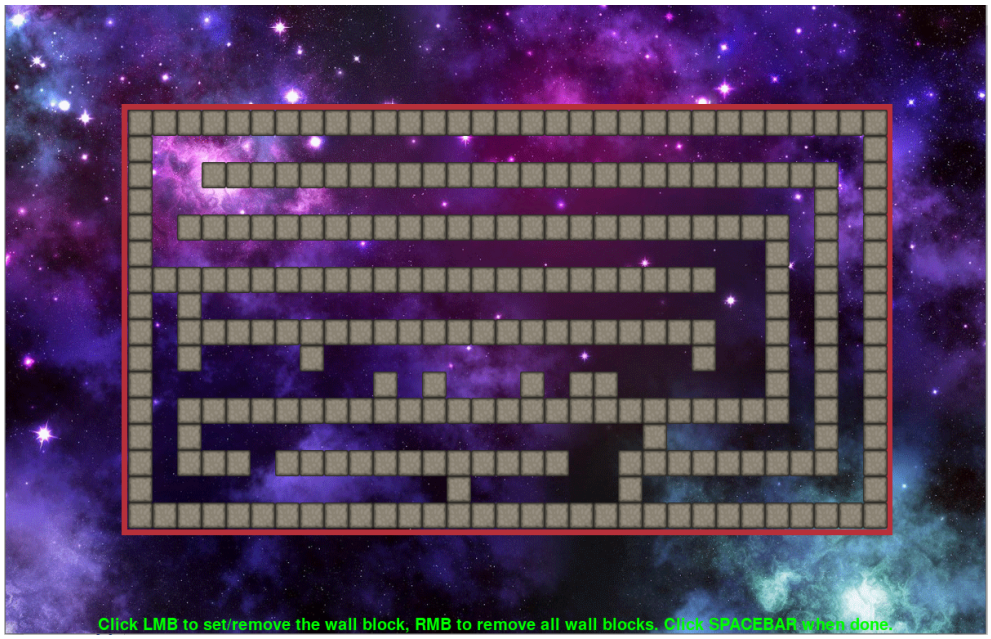

Figure 3.6: Maze no.2

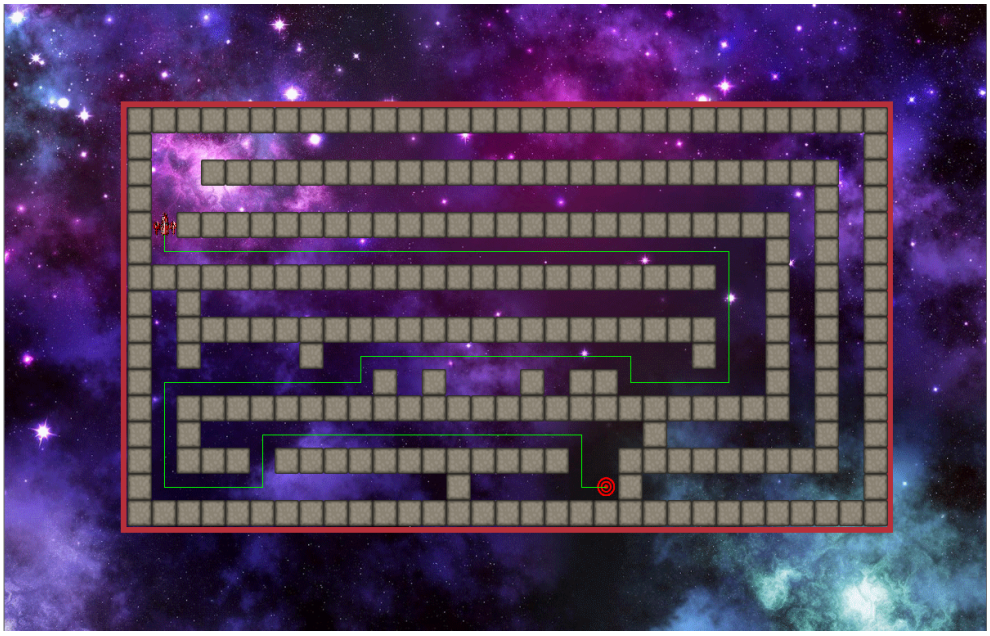

Figure 3.7: Maze no.2 solution

The third maze (see Figure 3.8), is the only baseline maze that expands outwith the red centre box. This was done purposefully to ensure symmetry. The idea behind this maze was to give the agent two paths with the same distance, thus same cost and see which one they would prefer. As can be seen in the solution in Figure 3.9, the unchanged A* prefers to travel downwards at every run.

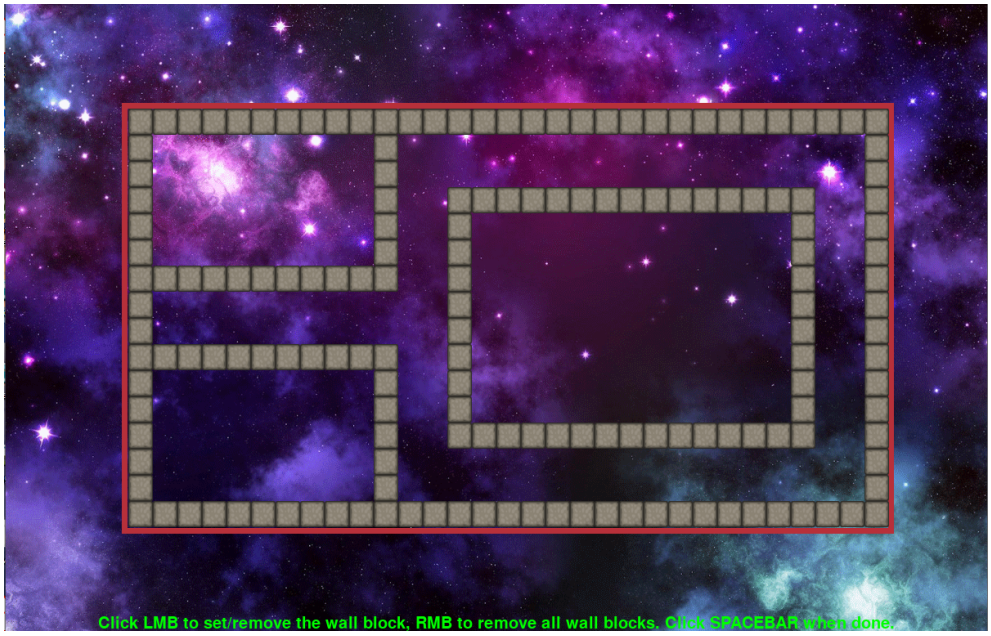

Figure 3.8: Maze no.3

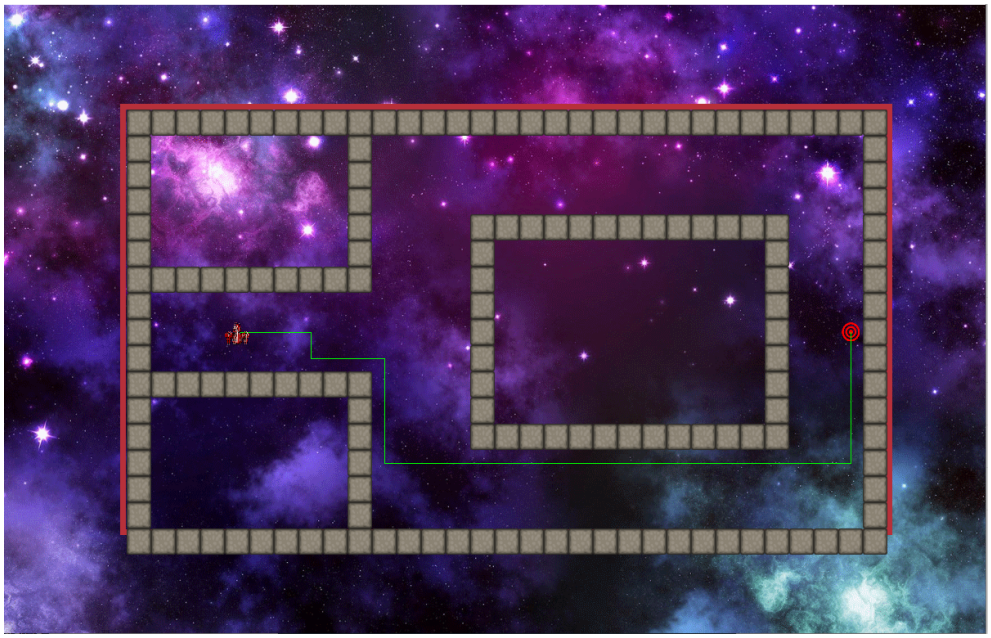

Figure 3.9: Maze no.3 solution

The fourth and fifth maze (Figure 3.10 and Figure 3.12) were designed to be the most complex and convoluted mazes out of the bunch. They include a large amount of paths the agent can take, while also including many cul-de-sacs too. The solutions to both can be found in Figure 3.11 and Figure 3.13.

Figure 3.10: Maze no.4

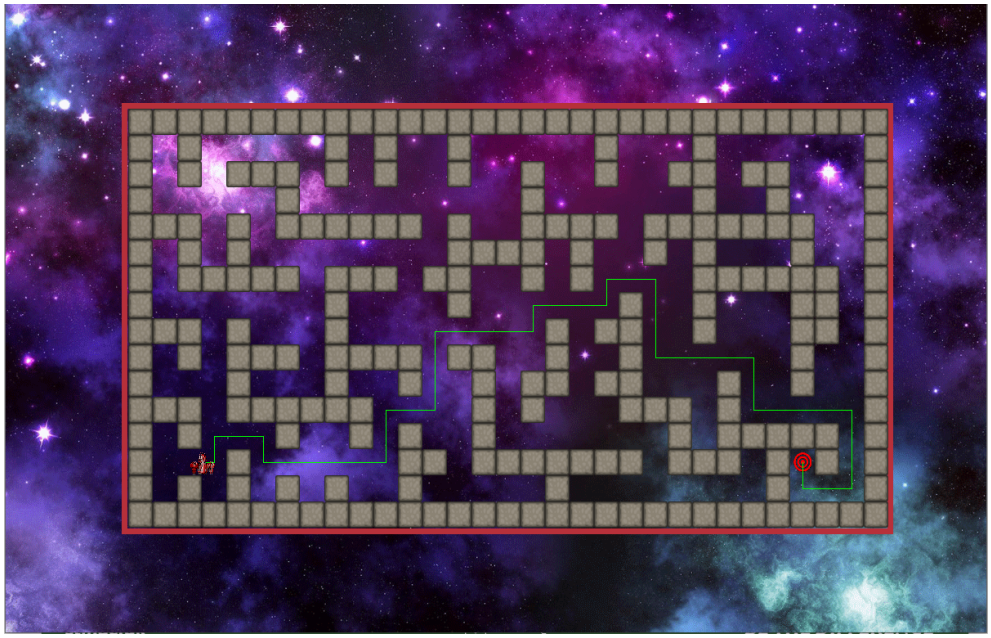

Figure 3.11: Maze no.4 solution

Figure 3.12: Maze no.5

Figure 3.13: Maze no.5 solution

The above shown mazes were then used to record the baseline values for the original A* algorithm before any improvements are made to it. All values using the improved algorithms are then compared against these. Due to background noise that will interfere with the times, all base metrics were recorded 10 times and then averaged out to form an average number for each maze. However, please note that most recorded time values were very similar and only had millisecond differences. These baseline numbers can be found in the table below, with the time represented in seconds and rounded to four decimal places.

Maze	Average time (in seconds)
N. 1	0.2796
N. 2	0.2419
N. 3	0.3131
N. 4	0.2498
N. 5	0.2989

Another important note to make before proceeding is that all algorithms used from here on out will have the same cost for every step. This means that every step the algorithm analyses will have equal costs and no node will cost more to be travelled to.

4 Improving the algorithm

4.1 Improving the A* using weights

Firstly, the decision was made to attempt improving the speed of the algorithm using the weighted A*. As mentioned prior, it is a popular method of improving search algorithms, frequently used in fields such as robotics and more. The interest here is to investigate the impact this particular algorithm could have in a video game context. It is known to significantly increase the speed by trading off optimality of the algorithm. The goal here is to analyse whether such a trade-off can work for video games too, since so far this has not been researched much. Video games are generally not as computationally demanding as robotics or complex satellite navigation systems. Therefore, the question arises whether the weighted A* can make a significant time improvement here on a much smaller scale, considering the baseline processing times are most likely already significantly lower than in more complex games for example.

4.1.1 Background on the weighted A*

While the A* works by following the function $f = g * h$, the weighted A* adjusts the heuristic component as follows:

$$f = g * \varepsilon * h$$

If the weight is 1, nothing changes and the normal A* algorithm runs, however if the weight is 0, this will create Dijkstra's algorithm. Thus, for the weighted A*, $\varepsilon > 1$. In this case, ε represents a weight that is used to increase the heuristics value. This creates a bias towards the nodes that are closer to the goal node [21].

4.1.2 Adaptation of the weighted A*

The weighted A* was run with the weights 1.2, 2.5 and 3.5 to ensure a good spread of values. The weights of 2.5 and 3.5 are commonly used for the weighted A*, while the 1.2 weight was picked as it only presents a slight change to the original A* algorithm and it was of interest to see whether a small change could make a significant impact. Every weight was tested 10 times on every previously mentioned maze. As with the baseline values, this was to account for any computer background noise that may be affecting the values. The results for each maze were then averaged out to create an overall time for each maze.

The mazes used were the same ones as for the baseline values to ensure a fair comparison (see Figure 3.4, Figure 3.6, Figure 3.8, Figure 3.10 and Figure 3.12). The values can now be seen along with the average non-adjusted values in the table below. The average times for all mazes are given in seconds and rounded to four decimal places.

Maze	Baseline	Average time (1.2)	Average time (2.5)	Average time (3.5)
N. 1	0.2796	0.2933	0.2674	0.2862
N. 2	0.2419	0.2382	0.1556	0.1606
N. 3	0.3131	0.2407	0.2357	0.2467
N. 4	0.2498	0.2337	0.2262	0.2412
N. 5	0.2989	0.2652	0.2587	0.2567

4.1.3 Results after implementation of the weighted A*

As can be seen in the table above, the results of this improvement attempt appear to be not very significant. Firstly, the algorithm using the 1.2 weight performed worse for the first maze than the original A*. However, this could be linked to background noise interfering with the values, meaning they would normally be similar in time. The other values for that algorithm seem to be slight improvements. Yet again, it is quite difficult to confirm this due to background noise. For the algorithm that used a weighting of 2.5, the averages seem to be slightly better across the board with the value for maze 2 being particularly good. And for the last algorithm with a weight of 3.5, the values start to rise slightly again. However, it still performed better than the 1.2 weighting. When it comes to the quality of the path, all the path solutions found by the weighted A* algorithm were identical to the original A* (see Figure 3.5, Figure 3.7, Figure 3.9, Figure 3.11 and Figure 3.13). Therefore, no optimality was lost when implementing the weighted A*. Overall, the results of the weighted A* were either not significant or not impactful enough to be considered a satisfactory improvement of the original algorithm. The slight differences in values are not consistent enough and could be the by-product of computer noise affecting the speed times. Therefore, other adjustments had to be tested in order to improve the search times of the algorithm.

4.2 Improving the A* using heuristics adjustments

4.2.1 Improvement attempt using the maximum f-score and randomness

As the results of the weighted A* did not give any satisfactory results, other improvement attempts were made to increase the speed of the algorithm. As mentioned before, the A* chooses a path through heuristic estimates given by the function $f(n)$. The algorithm will choose the node with the lowest f-score from the open set in order to build the path to the goal node. In theory, this should have as a result the best possible path, should the heuristic value be the most optimal. Inspired by the weighted A*, the idea was to find another improvement to the overall algorithm process that could increase the search time. Therefore, it was attempted to change the way the algorithm chooses nodes to build a path. Since in normal fashion it will deliberately choose the node with the lowest

f-score, it would be interesting to find out what would happen if the algorithm was told to choose the nodes with the highest f-score. No past research could be found on that topic, most likely as it might seem paradoxical to attempt this.

Additionally, it is important to note that during the recording of the baseline metrics, it was discovered that paths were always the same, no matter how many times the algorithm was run. This means that overall, the algorithm seems to lack randomness in the way it creates paths. Therefore it would be interesting to investigate whether added randomness could improve performance speed. It is often the case that randomisation can help increase speed in an algorithm. Therefore, a second improvement was made where the algorithm would choose a f-score randomly from the open set.

Therefore, two improvement attempts will now be made onto the original A* algorithm. On the one hand, it will be investigated whether added randomness can aid in increasing the speed of the algorithm. And on the other hand, whether letting the algorithm choose the max f-score can also affect the search speed.

4.2.2 Implementation of the new changes

Firstly, the algorithm was adjusted by letting it choose a random value from the open set. This means that instead of choosing the lowest value from it, it picked one at random, which was then used to determine which node should be travelled to next. Then, in a second file, the original algorithm was altered once more, but this time by making it pick the highest f-score out of the open set. Both new algorithms were then tested on the five previously seen baseline mazes (see Figure 3.4, Figure 3.6, Figure 3.8, Figure 3.10 and Figure 3.12). For each maze, each algorithm was run 10 times with the values then being averaged. Find below the table of these averages given in seconds and rounded to four decimal places.

Maze	Baseline	Average time (max)	Average time (random)
N. 1	0.2796	0.1611	0.2523
N. 2	0.2419	0.1782	0.2445
N. 3	0.3131	0.0722	0.2607
N. 4	0.2498	0.1059	0.2565
N. 5	0.2990	0.1447	0.2877

4.2.3 Results after implementing the two improved algorithms

As can be seen in the table above, the random algorithm showed slight improvements to the search speed times, similar to the weighted A*. Yet again, these slight improvements could be linked to computer noise affecting the times which is why it can not be confidently said that the algorithm is a true improvement. However, the algorithm using

the maximum f-score showed significant improvements across the board. The third algorithm especially seems to have been affected positively by this new algorithm.

However, as with the weighted A*, it is important to analyse the quality of the path as well, since generally, increase in speed means the quality of path can suffer in a trade-off.

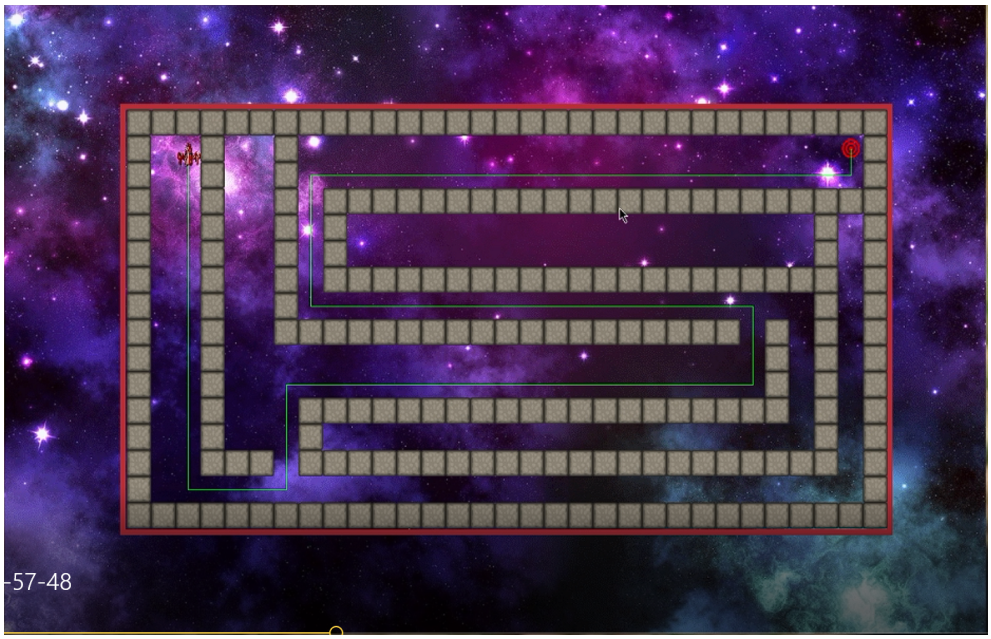

Figure 4.1: Improvement phase - maze 1 (max)

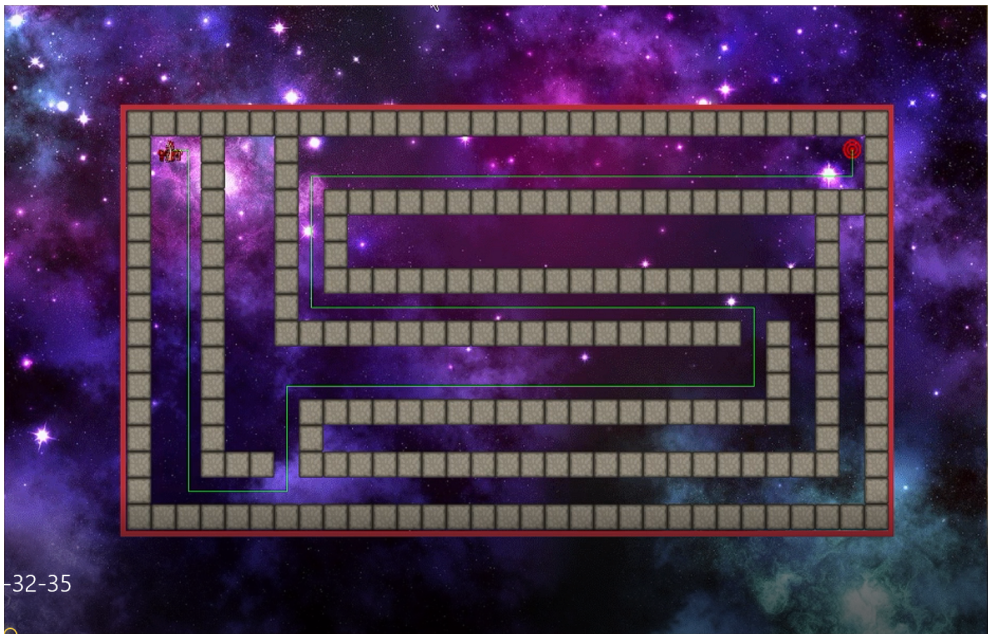

Figure 4.2: Improvement phase - maze 1 (random)

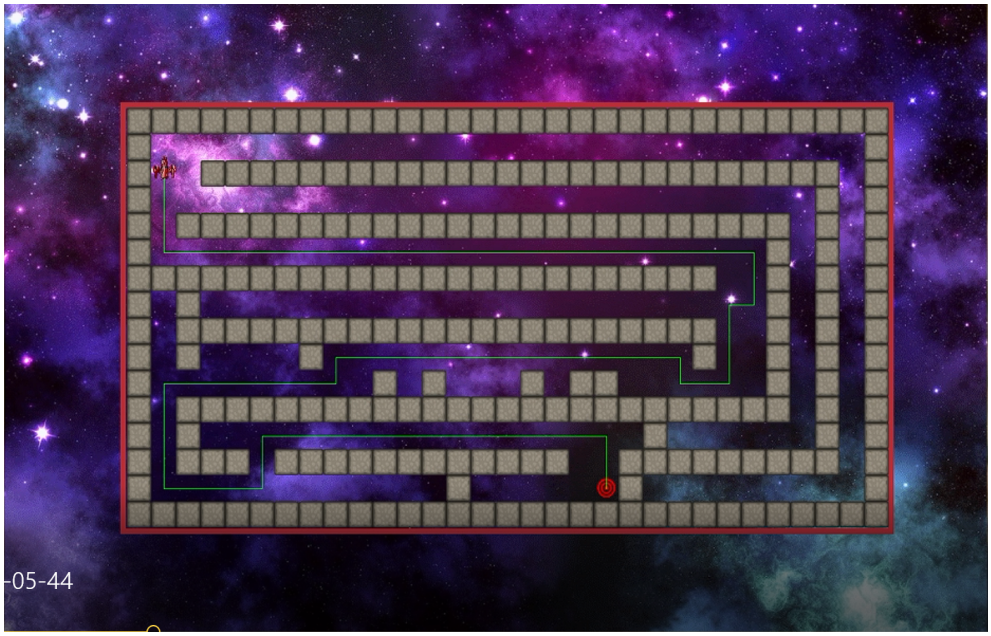

Figure 4.3: Improvement phase - maze 2 (max)

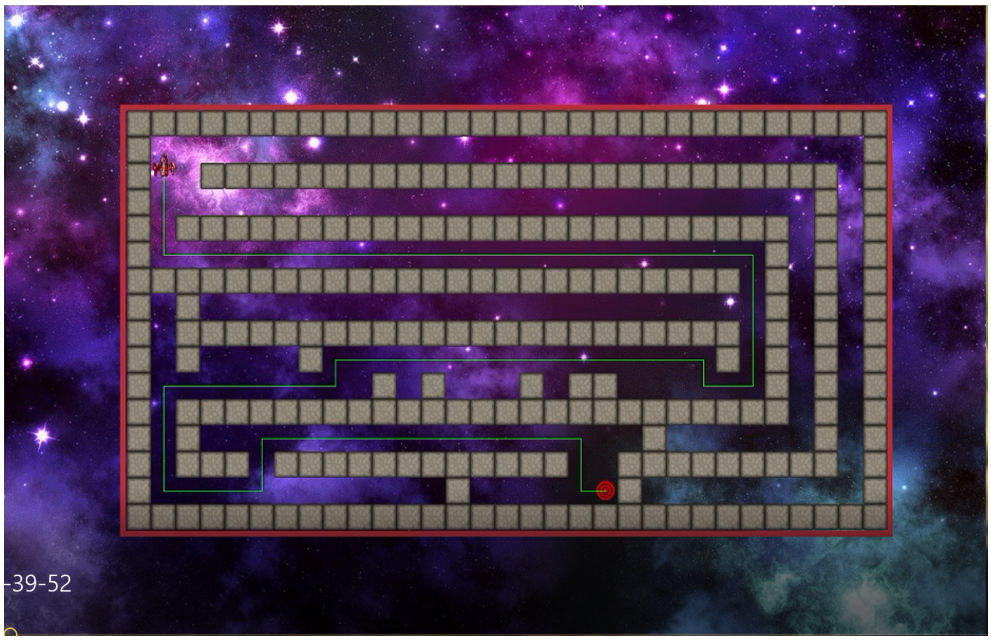

Figure 4.4: Improvement phase - maze 2 (random)

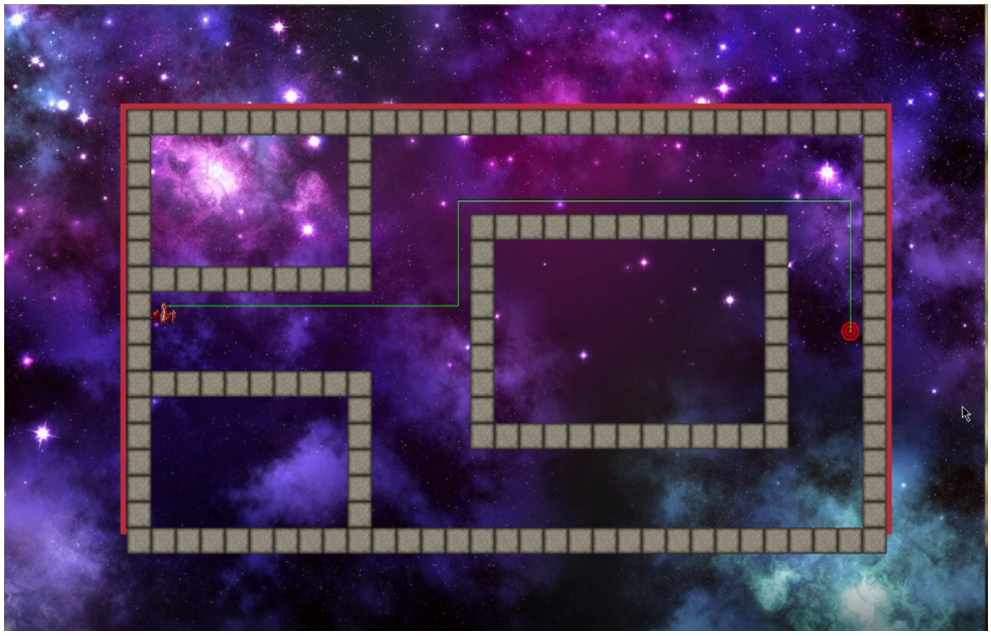

Figure 4.5: Improvement phase - maze 3 (max)

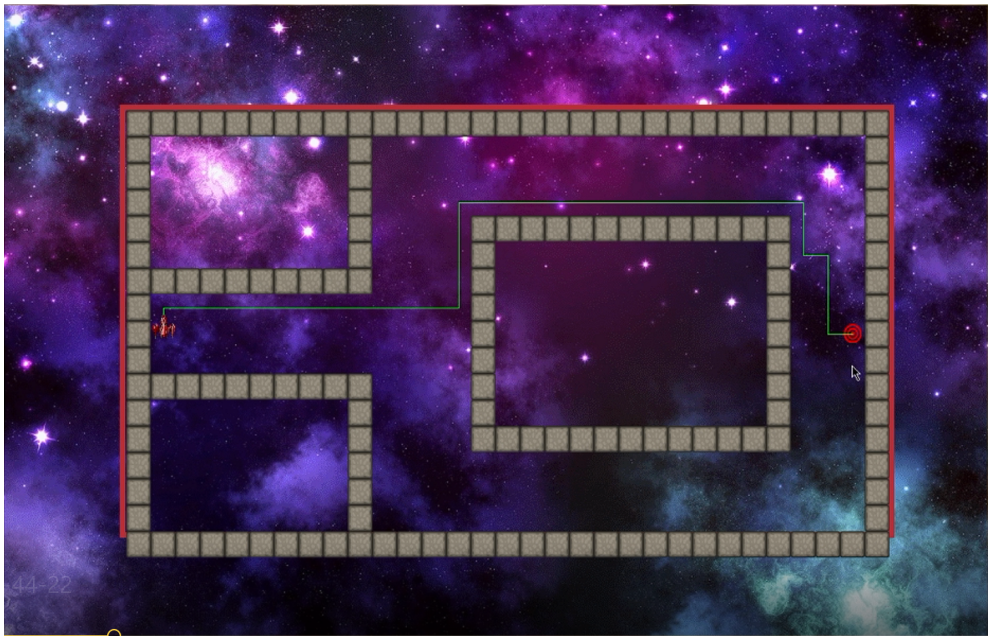

Figure 4.6: Improvement phase - maze 3 (random)

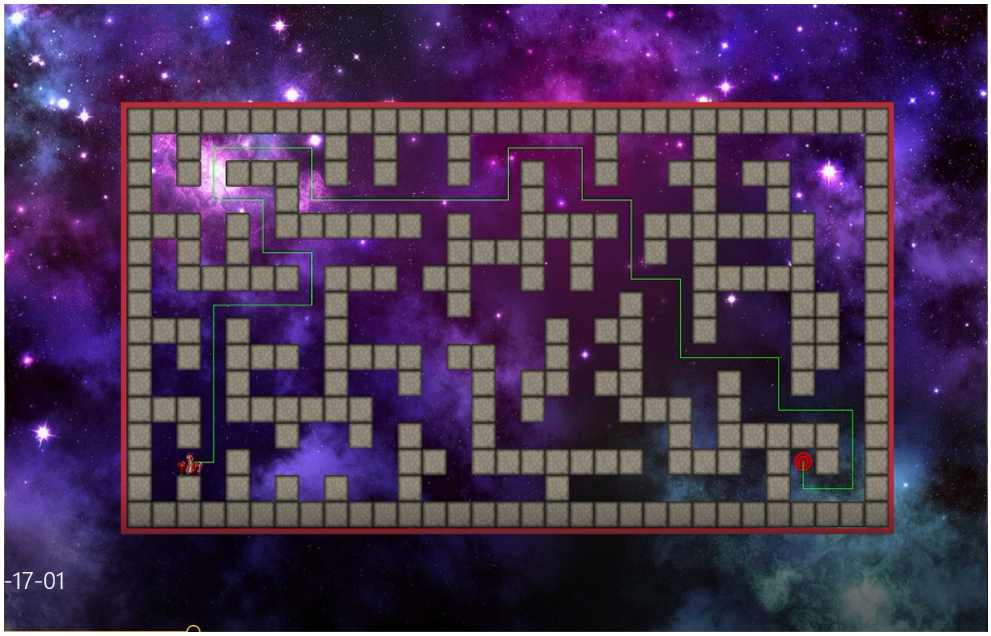

Figure 4.7: Improvement phase - maze 4 (max)

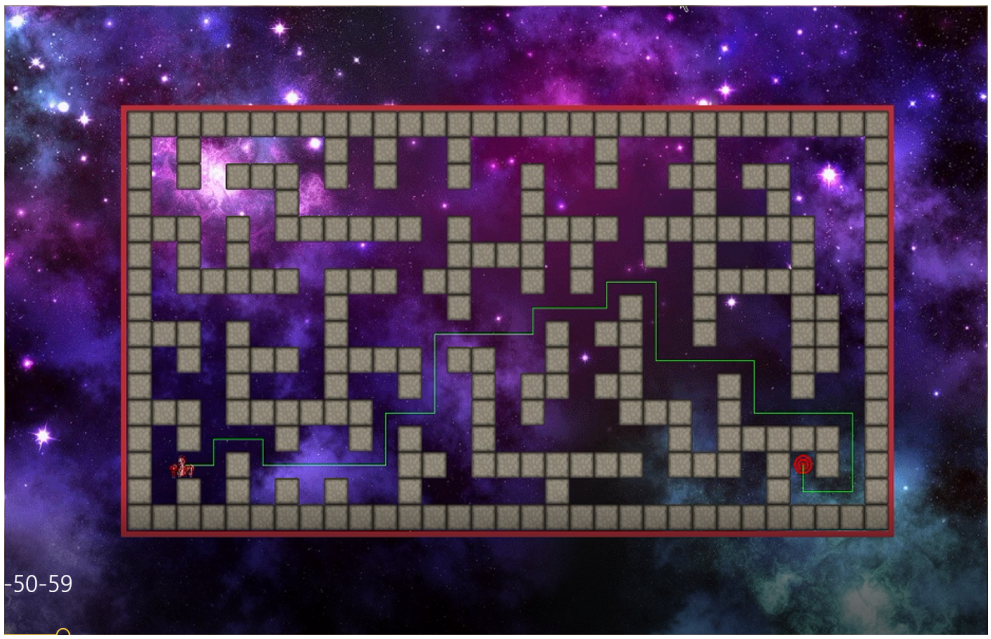

Figure 4.8: Improvement phase - maze 4 (random)

Figure 4.9: Improvement phase - maze 5 (max)

Figure 4.10: Improvement phase - maze 5 (random)

As can be seen from the figures above, there are some variations between the two new algorithms themselves, as well as the baseline solutions. These changes are not much of a surprise since the method with which the algorithm chooses to create paths was greatly altered. However, in the case of these five mazes, the proposed new paths do not seem to lose much in quality. There appear to be extra steps in a few paths, but

none completely wander off-course and create long travel times. A player would most likely not notice these differences. Thus, in the case of these mazes, the algorithms are adequate improvement suggestions.

4.3 Testing the improvements using random mazes

4.3.1 Comparison of the optimality of the paths

The first maze was simply the map without any obstacles, with the start node and the goal node being positioned at opposing corners of the screen (see Figures 4.11, 4.12, 4.13 and 4.14).

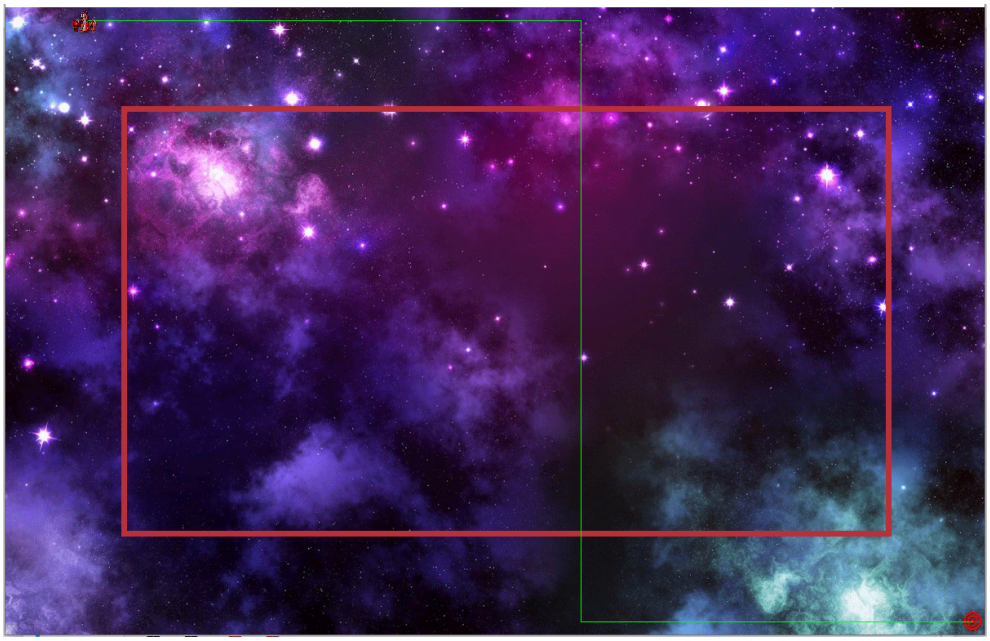

Figure 4.11: Testing phase - maze 1 (normal)

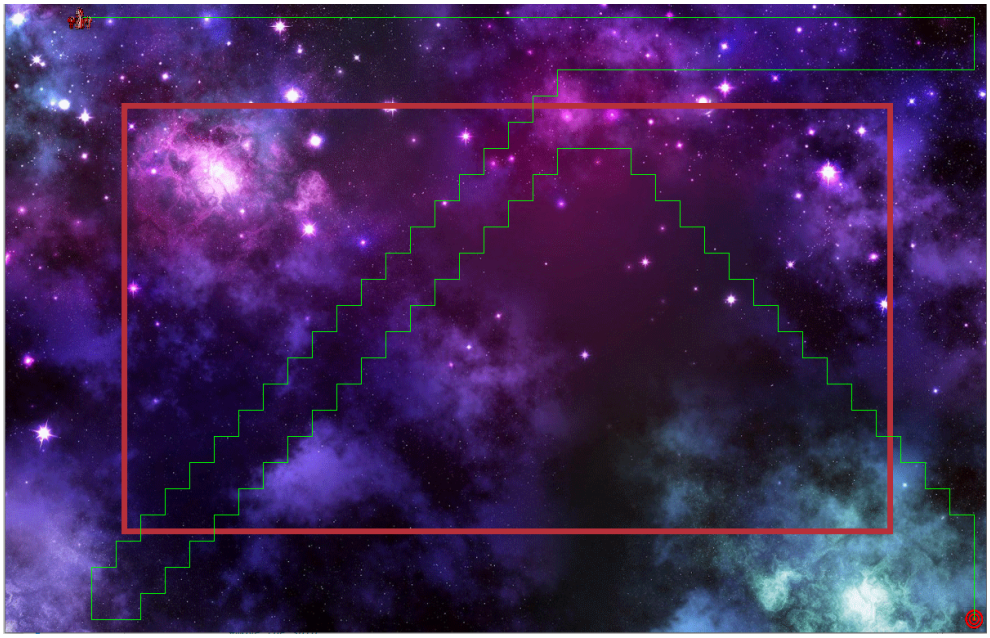

Figure 4.12: Testing phase - maze 1 (max)

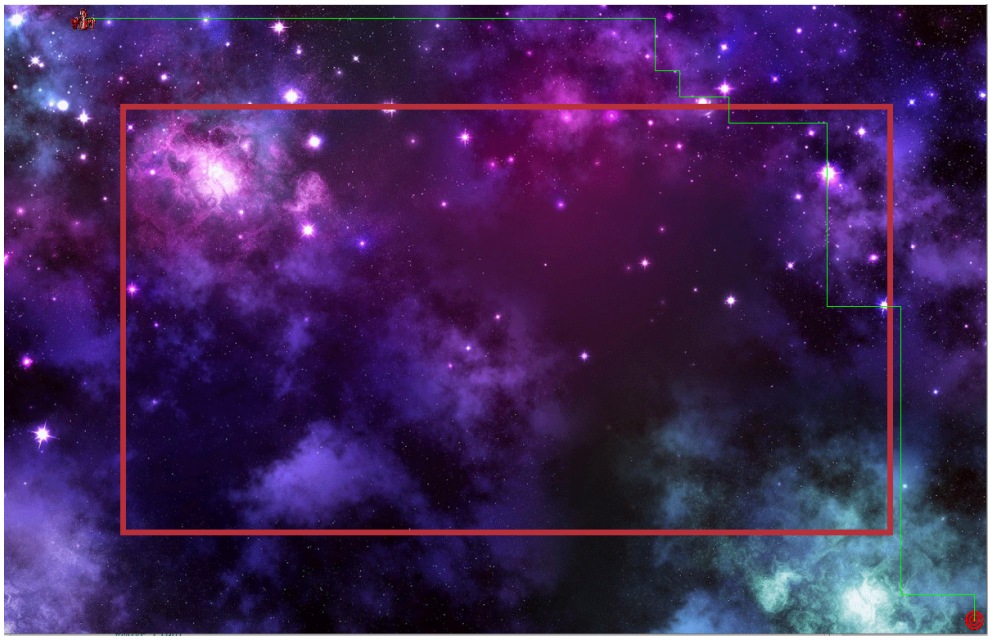

Figure 4.13: Testing phase - maze 1 (random)

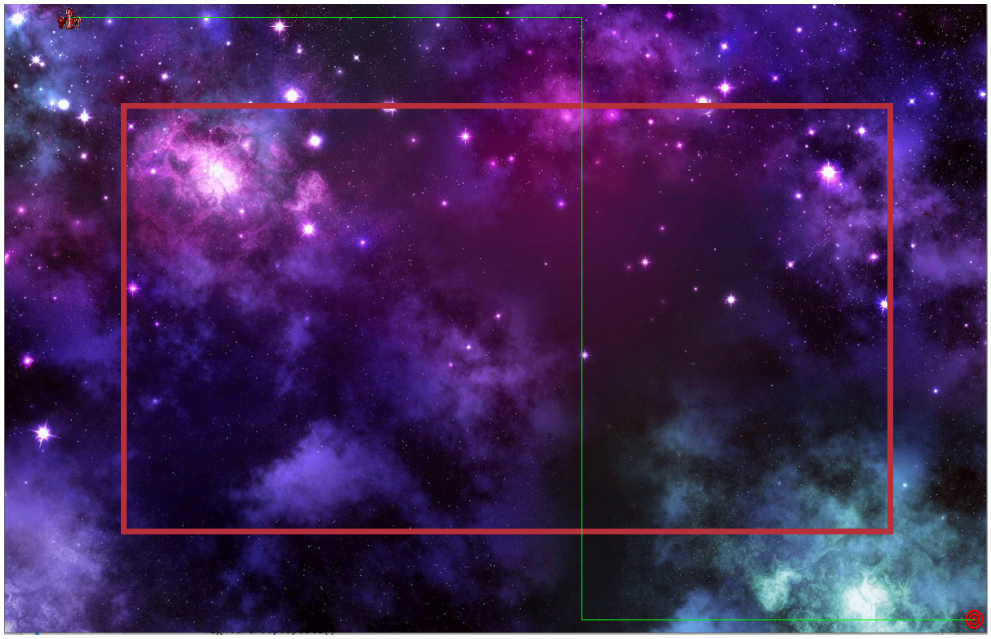

Figure 4.14: Testing phase - maze 1 (weighted)

As can be seen, the weighted A* performed the best in terms of path quality. The random algorithm had a few deviations, but nothing major that a player would notice. However, the maximum f-score algorithm went completely off-course by travelling across the map in a series of sharp turns.

The second maze was built using the same wall references than in the original mazes, however the obstacles were all laid out in a grid like pattern (see Figures 4.15, 4.16, 4.17 and 4.18).

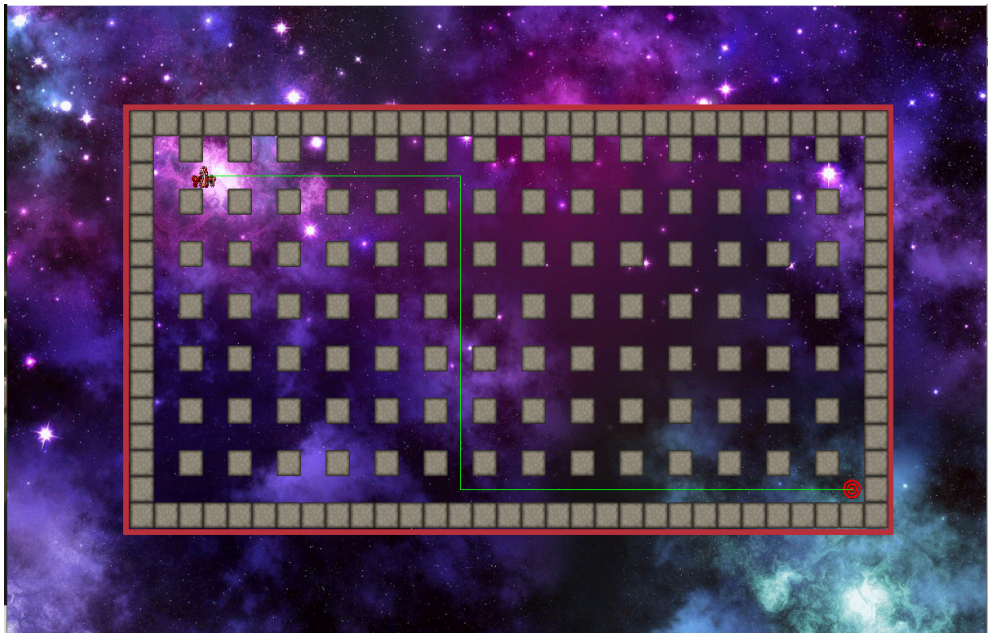

Figure 4.15: Testing phase - maze 2 (normal)

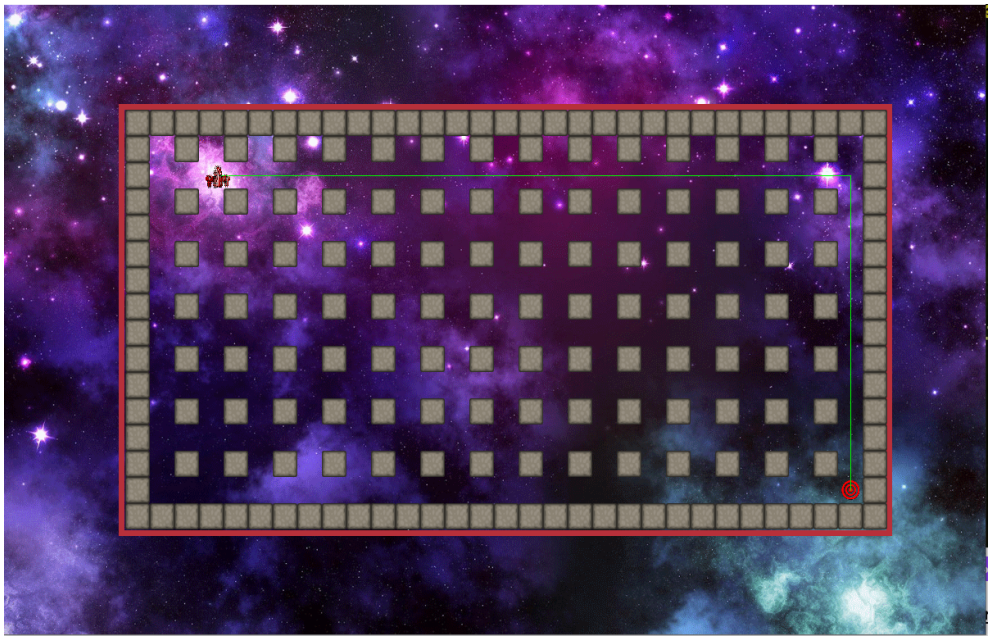

Figure 4.16: Testing phase - maze 2 (max)

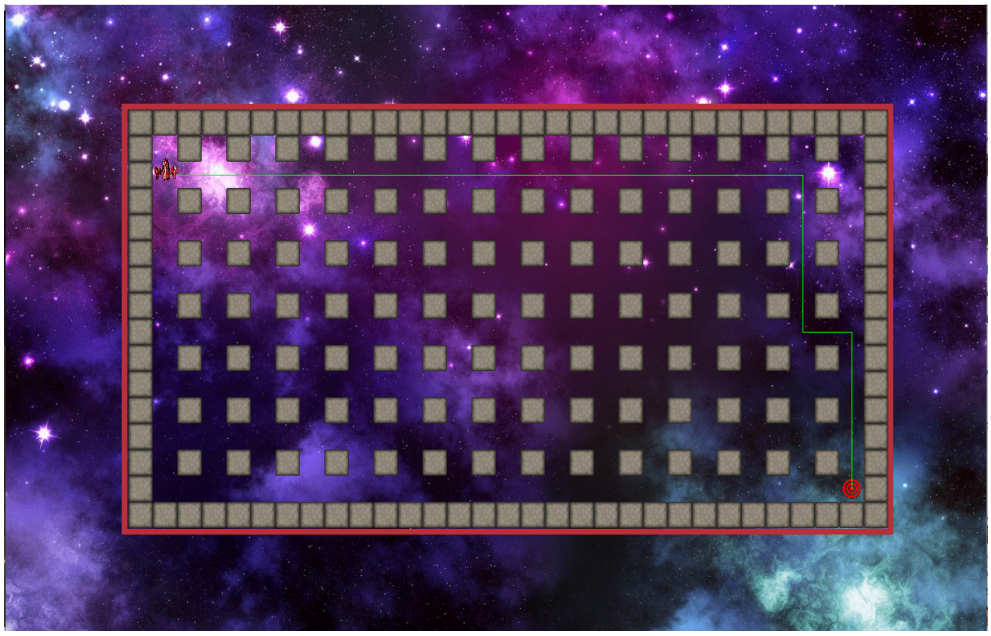

Figure 4.17: Testing phase - maze 2 (random)

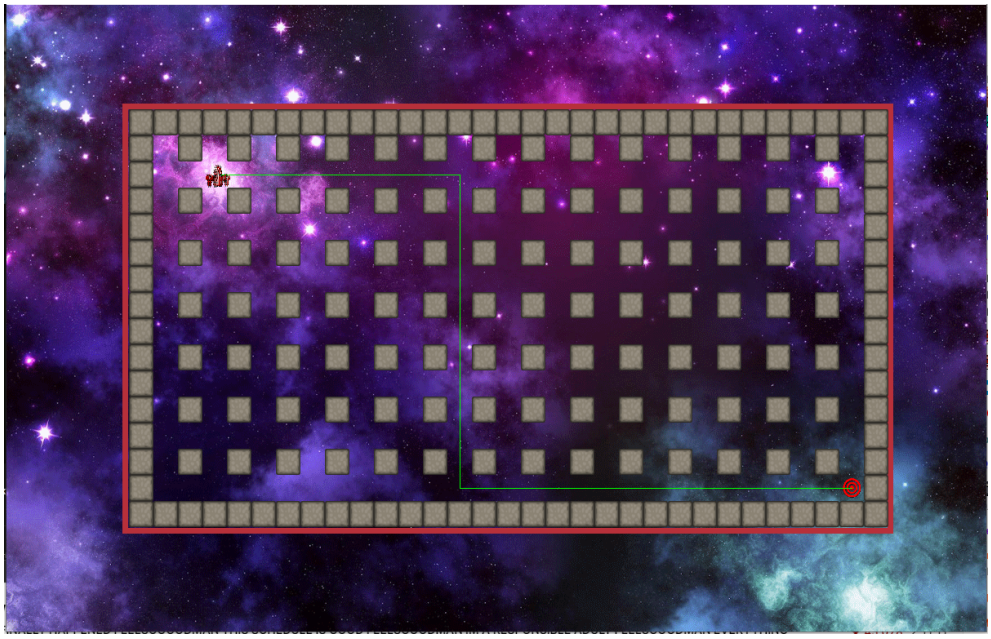

Figure 4.18: Testing phase - maze 2 (weighted)

In this configuration of the map, all algorithms seem to have performed adequately. The weighted A* once again creates the same path as the original, while the random algorithm travels further to the right of the maze. The maximum f-score algorithm seems to travel completely to the right, being close to the wall and only taking one turn.

Finally, the third maze used the same open concept as maze 1 however, the start and goal

node were separated by a dotted line of blocks (see Figures 4.19, 4.20, 4.21 and 4.22).

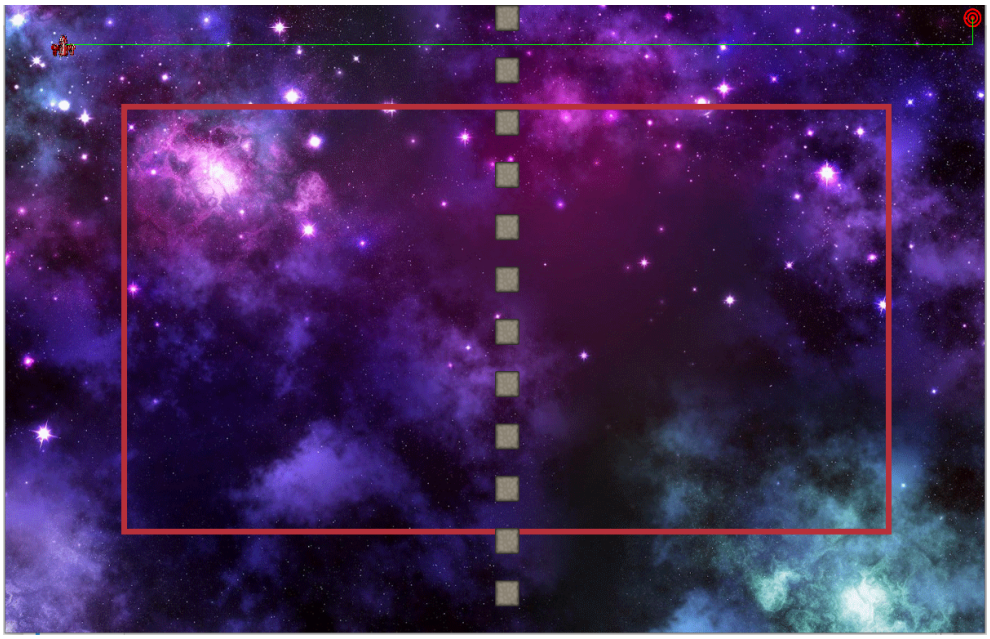

Figure 4.19: Testing phase - maze 3 (normal)

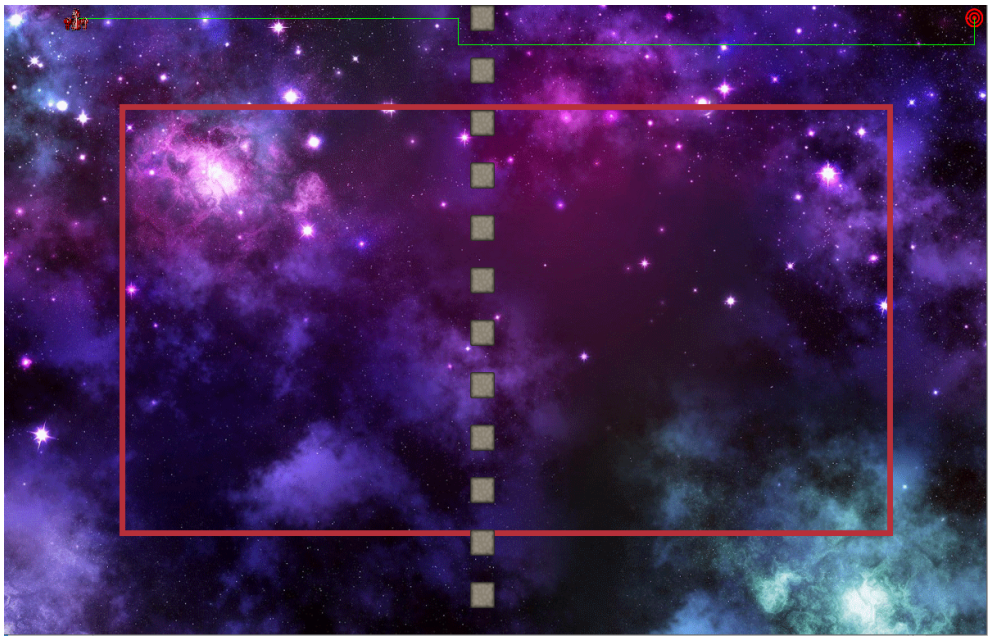

Figure 4.20: Testing phase - maze 3 (max)

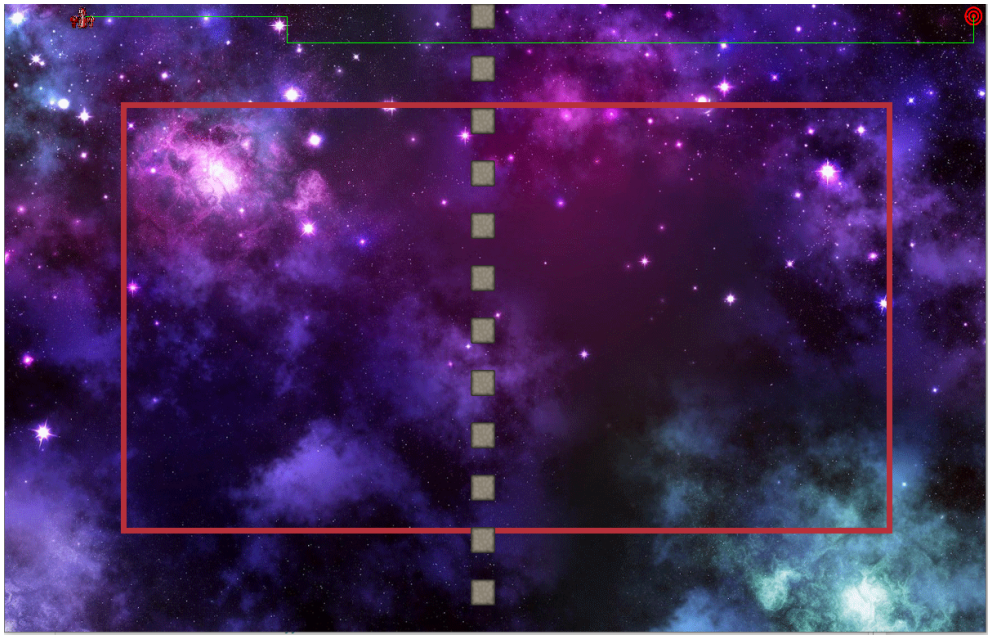

Figure 4.21: Testing phase - maze 3 (random)

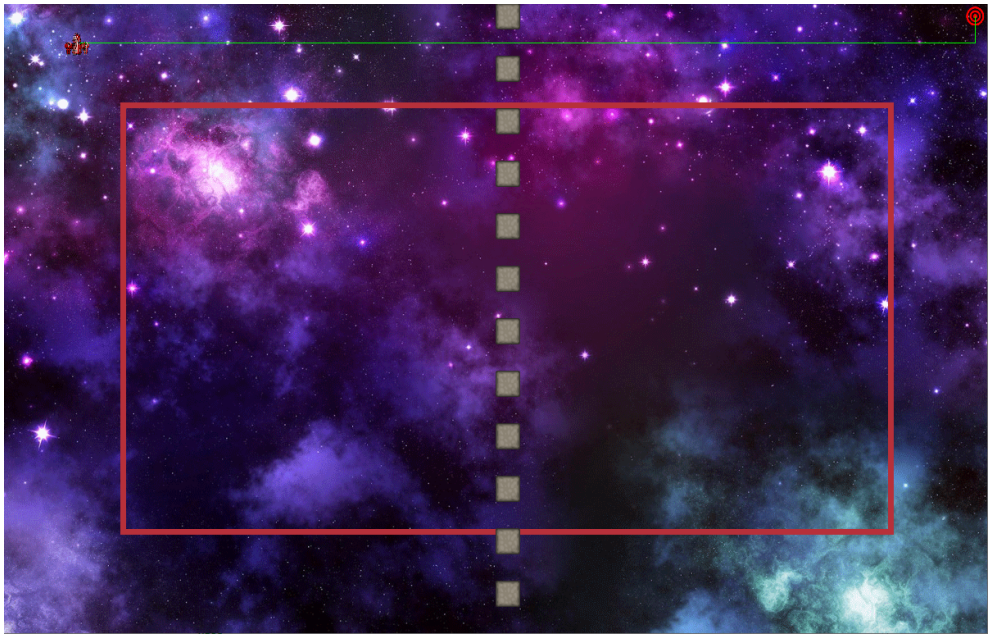

Figure 4.22: Testing phase - maze 3 (weighted)

Finally, here all four algorithms seem to have performed very similarly with the random algorithm and the maximum f-score algorithm only adding one turn to the path. Once again, these would be changes that a player would not mind and therefore the paths are adequate.

4.3.2 Results of the final testing phase

Now that the optimality of the paths has been analysed, the speed of the algorithms in the testing phase will be discussed. First, it is important to note that since the weights in the original tests did not give conclusive results, only one iteration of the weighted A* was included. The weighting was set to 2.5 as it seemed to have performed the best out of the three proposed weights. The time comparisons for the base algorithm, the weighted A*, the random algorithm and the maximum f-score algorithm can be found in the table below, with the times being in seconds and rounded to four decimal places.

Maze	Original A*	Average time (max)	Average Time (random)	Average time (weights)
N. 1	7.3633	6.5181	7.3828	1.0159
N. 2	0.5184	0.1321	0.5183	0.2957
N. 3	7.0826	0.03402	4.0009	7.1316

What can be seen from these numbers is that for maze 1, the weighted A* surprisingly performed the best out of all algorithms, followed by the maximum f-score algorithm. The random algorithm seems to perform slightly worse than the original A*. It seems that for the second maze, the random algorithm was also outperformed by the other two, with the maximum f-score algorithm performing slightly better than the weighted A*. Finally, in the third maze, the random algorithm seems to have significantly improved the search time, while the weighted A* seems to not have had much impact. However, the maximum f-score algorithm has made an immense improvement here once again.

These times show that the maximum f-score algorithm creates great improvements in terms of speed for some maps. However, as can be seen for the test maze 1, the quality of the path is simply not worth the time that is gained from using the maximum f-score. It seems that in the case of that algorithm, confined spaces are better suited since the option for paths is more limited. For the times where the maximum f-score algorithm seems to suffer, the weighted A* seems to be a great backup option, clearly outperforming it in the first maze. However, the random algorithm consistently stays behind in terms of speed, yet it seems to consistently give adequate paths. It appears to be a good middle-ground option should there be no room to run tests on the game.

5 Discussion and Conclusion

5.1 Discussion

The results from the first improvement attempt have shown that the weighted A* did not effectively work in this particular testing context. If there was an improvement in the processing times, then these could not have been detected effectively due to computer noise that affects the times as well. The general hypothesis as to why this was the case is that the game and search space might have been too small. A smaller search space led to a lower search time, resulting in the weighting not having a great impact. This does not mean that it should not be seen as a viable option to improve pathfinding. From what was seen during the testing phase, the algorithm could improve times over the proposed algorithm and the original A*, as was seen in the test map 1 (see Figures 4.13 and 4.14). It is most likely that open search spaces with less objects have more success with the weighted A*. However these are just speculations and more research should be conducted into finding out what factors affect the weighted A*.

Now, the second improvement phase tested two more algorithms, which also produced interesting yet mixed results. The random algorithm appears to somewhat affect work better than the original A*, however only during the testing phase for test maze 3 (see Figures 4.21 and 4.22) can a significant improvement be detected. It appears that here again, the map seems to have an influence on what algorithm will work best. Additionally, the maximum f-score algorithm seems to have produced some of the more impressive time differences, with its downside being the optimality and quality of the produced paths. The layout of the map, but also the location of the start and end node seem to play a role in the quality of the produced path. As can be seen in the test maze 1 (see Figures 4.11 and 4.12), when there are no obstacles and the start and end node are in opposing corners, the produced path is completely distorted. However, the test map 3 (see Figures 4.19 and 4.20), which had little objects and the start and end node across from each other, seem to have worked perfectly with the maximum f-score algorithm. In that case, this is probably related to the position of the start and goal node. Since the agent does not have to travel trough a large portion of the map, the produced path will be more contained and less likely to wander off-course.

Therefore, the results from the random mazes in the testing phase show that the improvements cannot be applied to all types of search spaces and maps. As previously mentioned, it seems that especially open spaces with no obstacles cannot benefit from the algorithm with the maximum f-score algorithm chosen, but the weighted A* appeared very promising, with faster speed and the same optimal paths produced by the original A*. The results also provide a lot of information on how the algorithm works. It appears that when not restricted by objects or walls, the maximum f-score algorithm will always create a path that is too long and sub-optimal. However, when surrounded by a closed-off wall the trade-off between speed and optimality can be accepted, as in such cases the player would not notice that the agent is taking a slightly longer route. There-

fore it is of great importance to consider adapting the pathfinding algorithm to the type of map that is being used in the game. For instance, should the game have a more open and empty space, then it is worth exploring the weighted A* or the original A*. However, for more compact and walled-off spaces, the maximum f-score algorithm could give some impressive time performances, in comparison to the A* and the weighted A*. Should the game have a mixed bag of maps, the random algorithm could be an option, but also potentially implementing different algorithms for different spaces. This is only a viable option should the extra added layer of code for a greater variety of pathfinding algorithms not use too much CPU power, rendering the trade-off null. Thus, it is always important to run tests in the game in question before choosing a search algorithm. Then again, the question remains whether going through the trouble of implementing multiple search algorithms is worth spending extra time for the potentially only small amount of speed gained. In circumstances where the extra time and resources do not want to be invested, the original A* algorithm is still a great option.

Nevertheless, it cannot be said for certain what makes the maximum f-score algorithm this much faster than its counterparts. A theory is that the open set is arranged with the largest f-score on top, making the retrieval process faster. If this is the case, then future work could experiment with rearranging the open list differently to see if the maximum f-score algorithm still works. If it does not, then the open space should be arranged with the lowest f-score at the top to see if the original A* now also runs significantly faster.

Another important thing to note is the size of this particular game used. It could seem that the improvements made here are not very significant as they are only a few seconds or milliseconds apart. However, in a much larger game with a search space that is a few hundred times larger, these times will probably add up, making the results more noticeable in some circumstances. However, the best practice is never to assume and generalise results and expect them to work in a different context. For this reason, future researchers or even game developers are urged to investigate whether this improved algorithm can truly give much better time values in a much larger game and map. However, in larger search spaces there is always the possibility that the trade off between speed and optimality works less effectively for algorithms like the maximum f-score algorithm. It might be the case that it only works faster due to the small size of the game, leading to a shorter open set. On a larger search space the open set will also be much larger, which means the algorithm has to filter through a much larger amount of f-scores.

However, the main theory remains that the results could probably be replicated in more complex games. This Pygame-based video game works ultimately like any other 2D game out there. Every game, when stripped from detail, is simply a search map like a matrix containing agents that travel from one square A to another square B. Even the most complex games rely on this same principle, with only the type of search space differing from game to game.

The algorithm improvements that were achieved in this project could now have great real life applications. As mentioned before, the game-player experience and continuous

improvement of the gaming experience is what developers are working on constantly. Small but important improvements like faster running algorithms could greatly increase customer satisfaction. A smoother and faster running game will lead to a happier player that will further want to engage with that video game company and buy their future products. High customer satisfaction will then also lead to positive reviews and comments on social media platforms. Nowadays, forum spaces such as Reddit and video platforms like YouTube offer video game customers a space for criticism. Before the internet was as prominent, opinions and critiques on games could not be shared as easily with other players. Therefore also negative opinions or issues with the game were less likely to be noticed by the general consumer. This is not the case nowadays, with screenshots of bugs or game mechanics being shared with anyone, often leading to these games being ridiculed by a large crowd of people. Sometimes bad press can be used in the game's favour, a famous example being Cyberpunk 2077, a game that was highly anticipated but had many issues with bugs and glitches at release. It received a lot of ridicule from players which helped bring the game a lot of traction and ultimately more sales. However, it is safe to say that generally video game developers prefer to produce games that have good reviews and reputations. These good reputations can be maintained by improvements made to the foundations of the game, such as the search algorithms.

Video games are also becoming more advanced and complex, with many saying the future of gaming is in virtual reality (VR). As the quality of games also increases, the importance of a good gaming experience becomes more prominent. Players also begin to expect a certain level of quality from their games, such as innovative designs and the most realistic graphics. To keep up with this constant demand for improvement, game developers have to continuously try to improve all aspects of their game. This desire for improvement also comes from the fact that more players, especially younger generations, are becoming more knowledgeable on the technical aspects of games. With the customer base becoming more tech-savvy, small improvements to the game can be more easily noticed. However, this means that smaller bugs and issues are also noticed, providing more room for criticism. With the younger generations becoming more tech knowledgeable, more are starting to build their own PCs and learn more about technology in general. However, this is often also the customer base that cannot afford to buy the best components on the market. This was especially difficult during the COVID-19 pandemic, where computer part prices surged due to the high demand of younger people building PCs and a general shortage of computer parts. This customer base will then benefit from games that are less CPU and memory demanding while still wanting to enjoy a great gaming experience. This again showcases the importance of making small improvements to video games, such as the algorithms that were developed during this project.

5.2 Conclusion and scope for the future

To conclude, this project successfully managed to create improvement algorithms for the A* algorithm. The tests performed uncovered interesting information on the A* itself and

how map layouts can affect a particular algorithm and the search speed. For instance, the weighted A* did not produce any significant results with the baseline mazes, but had an important impact during the testing phase on the open map. The random algorithm showed similar results by producing significant improvements in the testing phase. The maximum f-score algorithm however continuously showed significant time improvements, with the testing phase uncovering that some maps yield suboptimal paths. Therefore, all algorithms could be used to improve the original A*, however it is important to note their limitations when choosing to use them in a video game.

Future researchers are urged to run more tests on the proposed algorithms. The testing phase of this particular project was limited to only three additional maps, however probably much more information on the algorithms could be uncovered through further testing. Also, future work could test these algorithms in a game of much larger scale as well. Overall, it is also encouraged that game developers test these improvements in their games, as the time improvements gained could not only benefit them but also greatly increase the satisfaction of their customers.

Bibliography

[1] Ter-Sarkisov, A. A Star Pathfinding GitHub repository, https://github.com/AlexTS1980/A-star-pathfinding, August 2015.

[2] Xueqiao, X. Pathfinding GitHub repository, http://qiao.github.io/PathFinding.js/visual/, July 2015.

[3] Witkowski, W. Videogames are a bigger industry than movies and North American sports combined, thanks to the pandemic, https://www.marketwatch.com/story/videogames-are-a-bigger-industry-than-sports-and-movies-combined-thanks-to- January 2021.

[4] Cui, X and Shi H. A*-based Pathfinding in Modern Computer Games, IJCSNS, 11(1):125-130, January 2011.

[5] Mathew, G. E. Direction Based Heuristic For Pathfinding In Video Games, Procedia Computer Science, 47(1):262-271, January 2011.

[6] Algfoor, Z.A., Sunar, M.S., Kolivand,H. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games, International Journal of Computer Games Technology, 2015.

[7] Adaixo, M. C. G. Influence Map-Based Pathfinding Algorithms in Video Games, Universidad da Beira Interior, June 2014.

[8] Anguelov, B. Video Game Pathfinding and Improvements to Discrete Search on Grid-based Maps, University of Pretoria, June 2011.

[9] Bakkes, S., Spronck, P. and Herik, J. Rapid and Reliable Adaptation of Video Game AI, IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, 1(2), June 2009.

[10] Ponsen, M., Spronck, P., Munoz-Avila, H. and Aha, D. W. Knowledge acquisition for adaptive game AI, Science of Computer Programming, 67, 59-75, 2007.

[11] Falk, M. Artificial Stupidity, University of Kent.

[12] Botea, A., Bouzy, B., Buro, M., Bauckhage, C. and Nau, D. Pathfinding in Games, Artificial and Computational Intelligence in Games, 6, 21-31, 2013.

[13] Lee, W., Lawrence, R. Fast Grid-based Path Finding for Video Games, Canadian Conference on Artificial Intelligence, 2013.

[14] Peng, J., Huang, Y. and Luo, G. Robot Path Planning Based on Improved A* Algorithm, CYBERNETICS AND INFORMATION TECHNOLOGIES, 15(2) 2015.

[15] Han, S., Ye, L. and Meng, W. Artificial Intelligence for Communications and Networks, China: May 2019.

[16] Cui, X. and Shi, H. Direction Oriented Pathfinding In Video Games, International Journal of Artificial Intelligence and Applications, October 2011.

[17] Krishnaswamy, N. Comparison of Efficiency in Pathfinding Algorithms in Game Development, DePaul University, June 2009.

[18] Wang, J., Wu, N., Zhao, W., Peng, F. and Lin, X. Empowering A□ Search Algorithms with Neural Networks for Personalized Route Recommendation, Conference on Knowledge Discovery and Data Mining, August 2019.

[19] Wenfeng, B, Lina, H. and Yungfeng, S. An Improved A * Algorithm in Path Planning, International Conference on Computer, Mechatronics, Control and Electronic Engineering, 2010.

[20] Zhadan, A. Artificial Intelligence Adaptation in Video Games, Linnaeus University, 2018.

[21] Likhachev, M. A* and Weighted A* Search, Carnegie Mellon University.

[22] Ebendt, R. and Drechsler, R. Weighted A* search – unifying view and application, Artificial Intelligence, 173, 2009.

[23] Kindermann, T. H., Cruse, H. and Dautenhahn, K. A fast, three-layer neural network for path finding, Computation in Neural Systems, 7, 1996.

[24] McCabe, H. and Sheridan, S. Neural Network for Real-time pathfinding in computer games, Computer Science, 2004.

[25] Rafiq, A., Asmawaty, A. K. and Ihsan, S. N. Pathfinding Algorithms in Game Development, IOP Conference Series: Materials Science and Engineering, 2020.

[26] Foudil, C., Djedi, N., Sanza, C. and Duthen, Y. Path Finding and Collision Avoidance in Crowd Simulation, Journal of Computing and Information Technology, 2009.

[27] Graham, R., McCabe, H. and Sheridan, S. Pathfinding in Computer Games, The ITB Journal, 4(2), 2003.

[28] Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I. and Postma, E. Adaptive game AI with dynamic scripting, Machine Learning, 63, 217-248 2006.