Final Report

Man vs Machine

Creating AI for a multiplayer strategy game

Author: Jack Corscadden

Supervisor: Frank C Langbein

Moderator: Matthew J. W. Morgan

Module: CM2303 – One Semester Individual Project – 40 Credits

Acknowledgements

I would like to acknowledge my supervisor, Dr Frank Langbein, for his help and guidance throughout the many stages of this project. It would not be what it is without his assistance.

I would also like to thank everyone else who has helped me, be it the online community with their assistance during my implementation or my friends and family who helped me by testing my game and providing valuable constructive criticism.

Abstract

The main aim of this project is to create a fun and interesting 2D, turn based strategy game with AI opponents included.

The AI opponents will use either the Minimax search algorithm or the Monte Carlo tree search algorithm to decide upon the best possible move and they must prove to be a good challenge to human players.

These different AI bots will compete with one another on the game board in order to determine which of them is more challenging and interesting to play against.

The game is implemented using the libGDX framework and programmed in Java.

Table of Contents

Table of Figures

1 - Introduction

Technophobia gained attention as a phenomenon in England during the Industrial Revolution. Back then the fear of technology revolved around the fact that working people were being replaced by machines that were faster, stronger, more efficient and cost less money. The fear of technology replacing people's jobs still exists today but that word, technophobia, has grown and taken on many new forms.

It's now a common trope in science fiction: it could be the machines in 'The Matrix' which have an overwhelming majority of the human race enslaved so they can be used as batteries or the Replicants in Blade Runner, robots that have gained sentience and want to know why they are being used as slaves. The very idea of machines surpassing us intellectually and usurping us as the dominant species on this planet has been around for years and is prevalent in numerous popular movies, TV shows and games.

In a sense, this idea has already become reality. The game Metal Gear Solid 2 was lauded for the ways the guards would clear out rooms and chase the player down in a realistic manner, it felt as if the computer based guards really were sentient and searching for you. When they found you, it felt like the computer had outsmarted you. Even if you were to open up a very basic chess application on your phone or on your PC, it will likely be able to beat you unless you are an expert at the game. Computers are already able to perform better than humans in certain arenas, they may be limited in their complexity as they only need to learn one set of rules, but the fact they can beat humans at their own games is something that would have at one point in time seemed impossible.

Everything that has been detailed above acts as the inspiration for this project: creating a game with an AI player that is good enough to beat a human player. The project aims to create a game that is fun to play as a multiplayer human vs human experience, or as a single player human vs bot experience. The bots included with the game must pose an interesting and challenging experience and they obviously must have enough skill to beat human opponents.

This project would be of interest not only anyone with interests similar to those laid out above, but also anyone interested in **search trees**, **machine learning** and **game development**.

The scope of the project is fairly wide. It has involved the design and implementation of a multiplayer 2D strategy game. This included creating the rules, units, maps, sprites, sound effects and everything else. The project also involved the creation of three different AI bots that a human player can compete

against. Two of these bots are based on a Minimax strategy (an easy and a hard version) while the third bot was based on a Monte Carlo tree search strategy.

Prior to undertaking this project, the author had no experience in game development and design. Although the primary focus of the project was to make an AI that can compete with and beat human players, the work that has gone into creating the stage for the AI to compete on is worth note too. The hours spent designing, implementing, testing and tweaking the game have been extremely challenging but also immensely satisfying for the developer and have allowed him to increase his knowledge in numerous different areas of computer science.

In addition to the time spent creating the game, the work that has been carried out on the meat of the project, the AI players, has also been incredibly rewarding. Planning and implementing these players has granted the author an in-depth understanding of their core principles, as well as the advantages and disadvantages of each. Although the author had some basic experience with search trees and A* search from previously completed coursework, they had only heard about Minimax via bookwork and their knowledge of Monte Carlo tree search was basically non-existent.

Considering their limited experience with most aspects of the project and the strict timeframe, it was a big challenge. Therefore it was important for them to stick to the order and structure formulated in the initial plan in order to finish it in time, they couldn't afford to get bogged down perfecting tiny details such as graphics, sound effects or ensuring the code was completely clean and tidy.

It is important to have a well-defined list of aims and objectives before starting a project, to help keep it grounded and concise. A list of these were formulated in the initial report and they have been included as **Appendix A** in this report as they will be referenced multiple times as a list of important outcomes.

The project has been a success, virtually every one of the aims and objectives have been hit and the final product is a fun little game with a couple of challenging opponents included.

The results that have been compiled from the project are as expected. Through the testing of the different AI players that have been created it is visible that the Monte Carlo player performs ever so slightly better than the strong Minimax player, while both the Monte Carlo and strong Minimax player trounce the weak Minimax player. Based on the theory behind both of the different search algorithms it was predicted that the Monte Carlo player would perform better since it has the potential to consider moves far further in the future than the Minimax player.

This report will go through the project in detail. After noting some important background information, it will discuss the design, implementation and results before talking about some potential future improvements and analysing what the author has learnt.

2 - Background

The study of game AI is extremely popular and seems to be growing in popularity with every passing year. Two of the most notable examples are; in 1997 when IBM's Deep Blue defeated the reigning chess world champion and in 2016 when AlphaGo beat one of the best human players around. Both of these are examples of game AI beating some of the very best human players in the world. Although the game created in this project is different from both Chess and Go, both of these examples are good places to draw from when designing the AI players for this project.

2.1 - Minimax

Deep Blue used the Minimax search method in combination with Alpha-Beta pruning [1] in order to win its historic victory. The Minimax method involves checking every single move that is possible in order to find which move will **minimise the maximum loss** that a player can suffer down the line. The problem with Minimax is that it is computationally difficult to simulate every single possible move in a game of chess because it has a high branching factor (a high number of potential moves every turn) so it is not feasible as an opponent for a human player, since a human player won't want to wait for an extraordinarily long period of time before their opponent does anything.

This issue is tackled by creating a board evaluation heuristic. If the AI player can analyse a board and return a rating (how good that specific configuration of chess pieces is for the player) then it does not need to simulate every single move possible until a game over state is reached for every branch. Instead it can simulate a fixed number of moves into the future and use the evaluation heuristic to provide a rating of that state, allowing it to decide on moves within a far more reasonable amount of time. The trade-off here is it makes the bot a far less skilled player since it does not have as much foresight as before.

Alpha-Beta pruning alleviates this issue somewhat by allowing the search method to discard branches that are less useful to the bot (less likely to win from) so less simulations will need to be carried out on the whole. This allows the depth (number of moves in the future that the bot can simulate) to be increased while still being able to make moves within a feasible time frame. This is one of the bots that will be implemented for the project, a bot that uses Minimax in addition to Alpha-Beta pruning to determine which move to make next [2].

2.2 - Monte Carlo tree search

AlphaGo did not use Minimax, instead it used Monte Carlo tree search to determine which move to make next [3]. While the move simulation in Minimax is completely thorough down to a chosen depth (every single move is simulated down to that depth), Monte Carlo tree search exploits the most promising nodes and uses a degree of randomness to search moves with a higher potential of success to a much deeper level.

MCTS selects the most promising nodes from the tree and simulates random playouts from these nodes. The result of these random playouts are recorded and used to update the tree which allows the promising node selection stage to be more accurate. This continues until a large tree is populated but unlike Minimax it will not contain every possible move to a certain level, instead there will be branches with much more moves simulated than the others because they are the most promising moves.

One of the benefits of MCTS is that it can be halted at any time and still yield results. Obviously the more time it is given to run, the more accurate its decision will be. This is advantageous for something like a game as it allows the developer to give the AI players a fixed time in which to make their move unlike Minimax in which the time taken to make a move may be inconsistent. Another advantage is that the asymmetric structure of the tree means more time is spent simulating promising moves and less time is spent simulating obviously bad ones which in principle means the moves decided upon will be better.

However MCTS is not without drawbacks. For games of great or even moderate complexity MCTS can struggle to find reasonable moves within a certain time frame as the size of the search space may mean the key nodes are not identified and exploited early enough. It can also be extremely time consuming to simulate random playouts until completion which will affect the quality of the results that will be returned [4]. Despite these potential drawbacks, it was also decided that a Monte Carlo tree search player would be included in the game.

2.3 - Game

Equally as important to both of the AI bots is the arena in which they compete. The game itself was thought of as a hybrid between chess and the classic strategy game *Worms* [5]. It was envisioned as having a game board similar to chess but with walls and corridors to traverse. Instead of units moving onto or over the same tile as an opponent to capture them, they will have weapons with various different ranges and strengths. Instead of being free to select which unit to move during their turn, it will be locked in a strict progression so the player must think ahead as they may not get the chance to move a unit to safety for another eight turns.

With this in mind, it was decided that a game engine package would need to be found that would help with the implementation since the developer had limited experience with game development and did not have the time to learn every minute detail in terms of rendering objects on the screen, playing sound effects, etc.

The author's strongest programming language is Java so he decided that would be the one which would be used. It is said that Java isn't the best language for games but since the game for this project was to be relatively simple compared to modern AAA games and the author would already be learning a lot of new things as they went along it made sense for a Java based framework to be used.

One of the most popular Java based game frameworks is libGDX [6]. It offers numerous features such as input handling (mouse and keyboard), audio playback, graphics rendering and native TMX tile map support. All of these features would be beneficial in the development of the project and the tile map support was particularly interesting as it would make the creation of different maps relatively painless.

Slick2D [7] is a set of tools and utilities that provide similar features to that of libGDX. However they are less beginner-friendly than libGDX and have much less documentation and community support. Slick2D is also a dead project: although it has all the features that the project would require, it hasn't been updated since 2015 and thus it might be difficult to get technical support if any was needed.

Another recommended framework was jMonkeyEngine [8]. It has extensive documentation and support as well as having a huge suite of exciting features. However, it is primarily a 3D game engine and many of the features included such as shadows and shaders would be overkill for the project.

After some deliberation and research it was decided that libGDX was the best option as it is well documented and seems entirely appropriate for the project without having too much extra features that won't be needed. libGDX creates a framework of Java classes for a programmer to develop within but besides that it is basically just like programming in Eclipse as normal so it will not interfere with

the underlying plans for the gameplay and inclusion of AI by imposing a strict structure on development.

The aim of this project is to create a **fun** game and to compare which of the two AI bots, the Minimax bot and the Monte Carlo bot, are better equipped to provide an **interesting** and **challenging** experience when included as opponents in that game.

In order to discover this it was necessary to create the game and implement a version of both of the AI bots so they could then compete fairly with one another to produce a set of results that can be used as a mark of which of them is a better player.

3 - Design

A good design is absolutely essential for any project. It helps the developers manage their time since they can spend less time asking questions such as ‘how should X work’ or ‘what do I need to focus on next’ and more time implementing code.

Before implementing the game, it was important to clearly define the rules and basic gameplay loop. Just as chess has its playing pieces and board, this game would need its units and its map. It was also important to design the AI opponents that would eventually make their way into the game as well as to come up with an idea of what the visual elements of the game would look like.

3.1 - Requirements

The idea of ‘requirements’ has already been tackled by the list of aims and objectives laid out in the initial report (stored in **Appendix A)**. However these primary requirements can be expanded upon to include a handful of secondary requirements:

Visual Information

The game should include some visual information which helps the user keep a track of the progress of a match at any given time. It’s important to display this information in an elegant way and not bury it behind menus or in hard to reach places.

It should be instantly apparent who’s turn it is and this will be achieved by including a coloured tile in the top left of the map which corresponds to the colour of the current player making their move.

It should also be easy for a user to understand what possible moves they can make next, this will be done by highlighting the unit which is selected for that turn and by displaying possible moves and possible attacks on the board.

It should be obvious what state each unit on the board is in, this will be achieved by displaying a health bar on each unit.

Feedback for player input

Actions performed by the user (moving a unit, firing a shot, skipping a go, making a selection in the menus) must be acknowledged by the application or else they will lack impact. This will be achieved by including a range of sound effects in the game to be played as various actions take place. The noise of a unit being wounded will tell the player clearly that an attack has been made, for example.

Waiting on the AI

The wait time for the AI players should not be too long. This will be easily managed with the Monte Carlo player since an exact wait time can be specified. The Minimax player may be harder to manage since ensuring the wait time for it is not too great will involve making it a weaker player (giving it less nodes to search).

An optimum wait time for the AI player is two seconds, but the maximum wait time should be no more than ten seconds (some specific scenarios may require more computation than others and may lead to unexpectedly long wait times).

It is important that AI moves are clearly broadcast to the player. Care will be taken to ensure the movement and attack selected by the AI player are displayed on the board properly and do not simply happen instantly.

Explanation of game rules

The game will come with a guide which will display the rules and state the abilities of the different units. This will take the form of a PDF game guide which can be launched from the main menu.

The game should be interesting and fun

A difficult requirement to measure, but the game should be interesting in so much that each match has the potential to be different and have a unique feel. The best way to achieve this is to include a variety of maps and a variety of units. Some maps will be more open and favour sniper based gameplay whereas others will be closed with lots of tight corridors.

The game should also be fun to play. The fun experienced while playing the game will come via the satisfaction felt when performing skilful moves and taking out enemy units. Part of this sense of fun will come from the fact the player will have earned it themselves through their own cunning and skill, the other part of it comes from the response that the game gives them in terms of satisfying and appropriate sound effects.

3.2 - Gameplay

A lot of the existing research into both Minimax and Monte Carlo tree search has been done on relatively simple board games. Common examples are tic-tac-toe, connect four and chess. Chess in particular is a good example for showing the limitations of Minimax as it has an extremely high number of potential moves at any one time.

Chess served as an inspiration for the game design in many ways. The design and rules of chess are simple and easy to understand but there is an underlying complexity to it, expert players really are that much better than novice players and they can be extremely intelligent in regards to setting traps for their opponents whilst avoiding the traps that have been set for them. This is what the author wished to achieve with their game: a game that is easy to learn but difficult to master.

Not only would this make the game more interesting in terms of human vs human matches, it would also give the inclusion of AI players a lot more purpose. AI players will be able to view the game at a much higher level than humans, studying the entire board and every move possible for numerous turns in the future and thus being able to spot advantageous moves that the human opponents would not. This would allow the creation of difficult AI opponents, ones that can beat human players by performing moves that are difficult to defend against because unexperienced human players would be unable to spot them.

With the above considerations in mind, a design was formulated for the game. It was decided that the game would be based around the following components:

A game map would serve as the arena in which matches would take place. The map would be a 2D tile based map with two types of tiles: floors and walls. The difference between these tiles are easy to understand, floors are open pathways whereas walls are barriers. These would be constructed in such a way that the map has different choke points and numerous ways for a player to move their units around the different areas.

A selection of units would sit on top of the game map. These units will belong to either player one or player two and will be of a specific unit type. The unit types will be fleshed out in a later section, but each unit type has different abilities meaning they can be used in different ways and for different strategies. These units would be able to **traverse** the game map and **attack** other units on the board.

Two players would compete in every match. Each player would take turns making moves and each move would be comprised of two phases. The first phase is the movement phase where they can move a pre-selected (via a fixed order) unit around the board, the second phase is the attack phase where

they can then attack with that same unit. Once a player has finished their move, the other player will make theirs and this will continue until there is a winner.

The above information is a succinct description of how the game will play. It can be used to craft an easily understandable gameplay loop:

1) **Player 1, Phase 1** – Select a tile on the board to move a unit to
2) **Player 1, Phase 2** – Select a tile on the board for the previously moved unit to attack
3) **Player 2, Phase 1** – Select a tile on the board to move a unit to
4) **Player 2, Phase 2** – Select a tile on the board for the previously moved unit to attack
5) **Repeat –** the above steps will be repeated and will be interrupted only when one of the players has lost all of their units

3.3 - Units

Variety is important in a game, it helps keep the gameplay interesting which is a requirement for this project. So it was decided that a variety of different unit types should be included. The units in this game are the moveable pieces available to each player and having different units behave and act differently is an excellent way to introduce the possibility of numerous tactics and playstyles.

Before discussing the ideas for unit types, it is important to completely define what a unit is in the context of the game:

A **unit** is a controllable entity within the game that has a set of attributes: type, movement range, attack range, health, strength and the ability to shoot over walls.

The **type** indicates which unit type the unit is.

The **movement range** indicates how far a unit can move during a turn.

The **attack range** indicates how far a unit can attack during a turn.

Health indicates how much damage the unit can withstand before dying.

Strength indicates the amount of damage the unit can inflict with a single attack.

The **ability to shoot over walls** determines if a unit can attack over walls and over other units.

There are other attributes attached to each unit such as team, current position and whether they are currently alive or not, but the above are the main ones that are relevant to this section of the report. From the above stats I have come up with three different unit types.

Rusher

The *rusher* is a swordsman. It is supposed to be fast-paced but with a short attack range. Similar to the pawn in a game of chess, the *rusher* isn't a great fighter but can be used effectively in blocking off opponent moves and protecting other units.

The *rusher* has a **movement range** of 5, an **attack range** of 1, a **health** of 1 and a **strength** of 1. It **cannot** attack over walls.

Flusher

The *flusher* is a grenadier. It only has a medium movement and attack range but it has the unique ability of being able to attack over walls. The *flusher* is fantastic at attacking defensive units as it can launch attacks right into an opponent's base. It is also has the greatest health of the units, being able to withstand two hits from *rushers* and other *flushers*.

The flusher has a **movement range** of 3, an **attack range** of 4, a **health** of 2 and a **strength** of 1. It **can** attack over walls.

Shusher

The *shusher* is a sniper. It has a small movement range, but an extremely large attack range. Although it cannot attack over walls, the *shusher* is able to hit units at the opposite end of the map. This fantastic range is supplemented with enormous strength as the *shusher* is able to take out an opposition unit in a single shot, no matter what unit type they are.

The *shusher* has a **movement range** of 2, an unlimited **attack range**, a **health** of 1 and a **strength** of 2. It **cannot** attack over walls.

3.4 - Interface

The interface in the game will be minimal. Only three different screens will be created; the title screen, the game screen and the results screen. It's still important to think about these beforehand as it will aid when it comes to implementing the screens in terms of what graphics/text to create and how to place them.

Title Screen

The title screen has a few necessary elements that must be included. Since its main use is to *set up the game* it will allow the user to select what type of players (human or AI) and which map will be used. It will also include an option to open and view a guide for the game.

Here is a mock-up of how this screen will look:

Figure 1: Title screen mock-up

The lighter grey patches signify clickable buttons. Hence in the above example, if the user wishes to play a game against the AI instead of against another human player, they will click the "HUMAN" button beside 'PLAYER 2' and it will change to a different player type.

Clicking the 'PLAY' button on the above screen will launch the game with the current configuration of players and map. Clicking the 'HELP' button will open a PDF guide of the game.

Game Screen

The list of additional requirements mentioned that the game should display important information in an elegant way so that a user is not burdened with trying to navigate a complicated or intrusive HUD. It is important that the vast majority of the game screen is dedicated to showing the state of play on the game board and is not cluttered with additional information.

Here is a mock-up of the game screen:

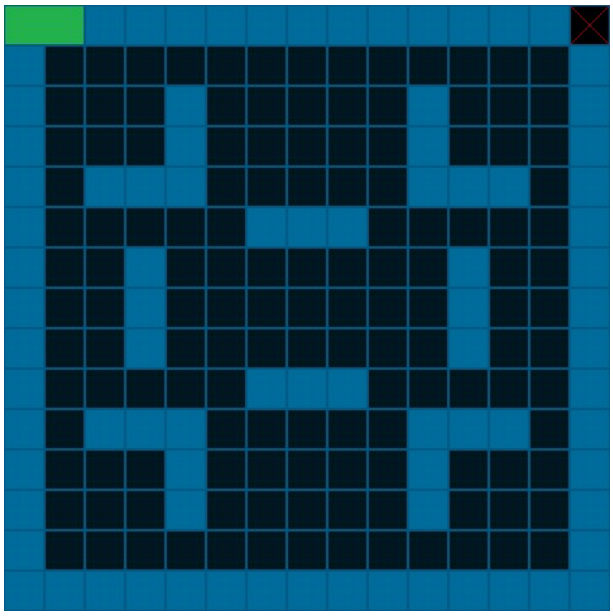

Figure 2: Game screen mock-up

The green rectangle in the top left corner will change depending on which player is currently making their move. Player 1 will be coloured red and thus when it is their turn the rectangle will be red. Player 2 will be green so when it is their move the rectangle will be green.

In the top right corner there is a clickable box that will quit the game and return to the menu.

The rest of the mock-up is just a visual representation of how the game will look. The grid is divided into two tile types: the lighter blue wall tiles and the darker blue floor tiles. Units will be able to move about this grid and will attempt to attack one another.

When a unit is selected, every tile that it can move to will be highlighted with a yellow outline. When the unit is moved to one of these tiles (the unit can also be made to stay in the same tile) then every tile that it can possibly attack will be highlighted with a red outline. Thus, all possible moves are displayed on the screen without needing to clutter it with unnecessary HUD information.

Results Screen

The results screen will be simple. It will display the identity of the winning player (along with their team colour).

Figure 3: Results screen mock-up

The entire screen will be clickable, and upon being clicked the user will be returned to the title screen where they can configure another match.

3.5 - Game Map

libGDX comes with an inbuilt ability to render TileMaps by loading .TMX files. This will allow maps to be created with ease using an application called **Tiled** which provides an easy WYSIWYG editor to design tiled maps. This makes the creation and loading of tile maps trivial.

3.6 - Opponent AI

One of the key components and deliverables of the project is having a working AI opponent to play the game with. As outlined in the list of aims and objectives earlier there will be more than one AI opponent and there will be a total of two different search algorithms used: Minimax and Monte Carlo tree search.

In the final deliverable of the game there will be three included AI players:

Max

Max will be a strong player that uses the Minimax algorithm to determine the best possible move. Minimax is complicated and there will be certain ways in which the algorithm must be tailored specifically for the game. This section will detail how the Minimax algorithm to be used by the AI players will work.

To begin with the game must be viewed as a tree. The current state of the game (the configuration of units on the game board, including their position and health) is the **root node** which we **expand** by creating and attaching child nodes. One child node will be created for each move that can be made by the currently selected unit in the parent node.

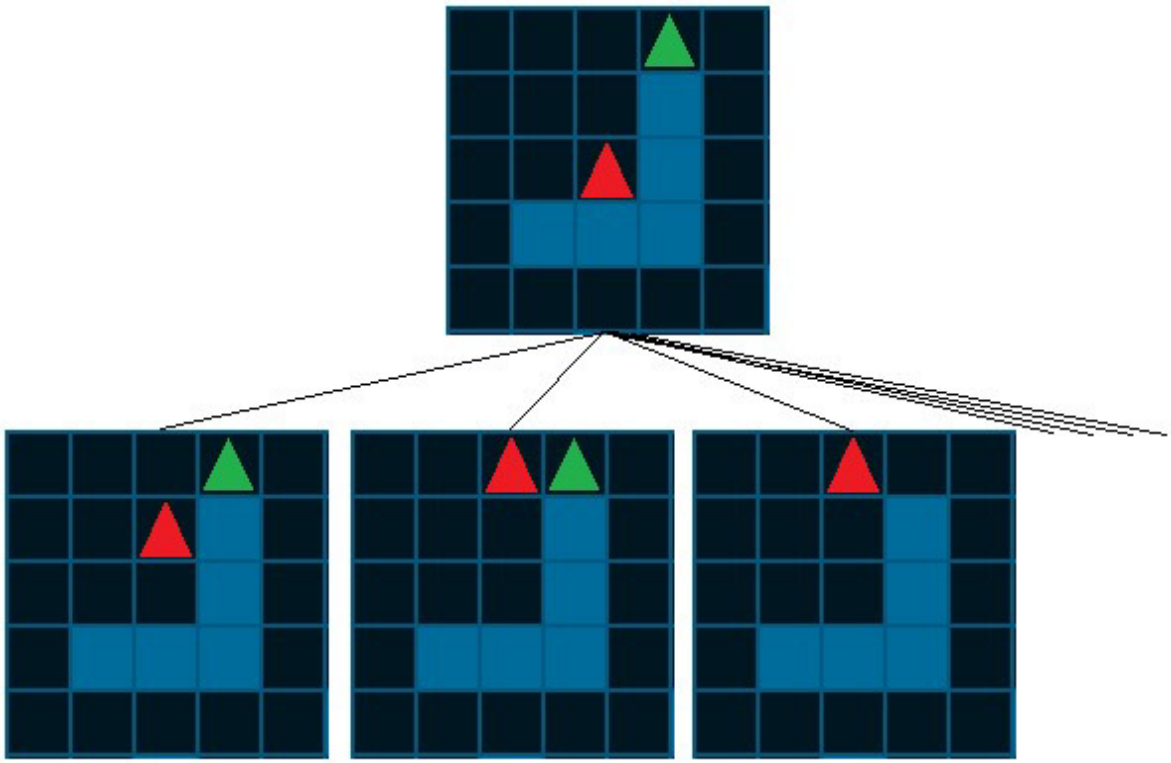

Figure 4: Basic Minimax tree, the selected unit in the root node is the red triangle

Each of these new child nodes will represent a potential move and thus will store the state of the game after that move is made, three of which are shown in *Figure 4* above. From each of these new nodes the process of expansion that was performed on the root node will be repeated: new child nodes for each possible move will be created. During this stage the simulating player will also be simulating the moves that are possible for the opponent player. This continues down to a fixed depth, for example **five moves** into the future by which time the AI player will have access to a branching tree structure, representing every single possible move down to that depth.

Now that the AI has access to a tree which holds every single state of play possible after five moves have been made, the next step is to carry out a heuristic evaluation on each of these final states (the leaf nodes at the bottom of the tree).

A board evaluation heuristic will be used. It will begin at a value of 0 and increase/decrease based on various factors. These values will be easy to change in the case that the game needs to be balanced, but for a basic idea of how they will work the following values are a good starting point:

- add a value of 500 for any surviving *Flusher* unit on the current players team
- add a value of 400 for any surviving *Shusher* unit on the current players team
- add a value of 200 for any surviving *Rusher* unit on the current players team
- subtract a value of 500 for any surviving *Flusher* unit on the opponent players team
- subtract a value of 400 for any surviving *Shusher* unit on the opponent players team
- subtract a value of 200 for any surviving *Rusher* unit on the opponent players team
- if the current player is dead, return the minimum possible value
- if the opponent player is dead, return the maximum possible value

Using these evaluation markers, an AI player will be able to determine if a simulated game state is good or bad.

If the simulating player has one of each unit alive while the opponent only has one *Rusher* alive, the evaluation function will return a value of 900. This is a favourable position because the simulating player has more surviving units, and stronger surviving unit types so a positive value of 900 reflects this well.

If the situation is flipped and the simulating player only has one *Rusher* alive whilst the opponent has one of each unit alive, the evaluation will return a value of negative 900. This is an unfavourable position because the simulating player has less surviving units, and the opponent has stronger

surviving unit types. Thus we can see that the evaluation function will give a good indication of good and bad game states.

So the AI player can create nodes that will simulate the game board for a number of moves into the future and it can work out how effective these moves are based on the evaluation of the final states, but how is this information used to make an informed decision on what move is the best? The simulating player, the **max**imising player, will assume that the opponent, the **min**imising player, will always play optimally and attempt moves which will return a **lower heuristic value**. So out of every possible move, the minimising player will choose one that disadvantages the simulating player the most (because this puts themselves at an advantage). The following figure will show how this works in a simplified manner (a depth of two moves in the future):

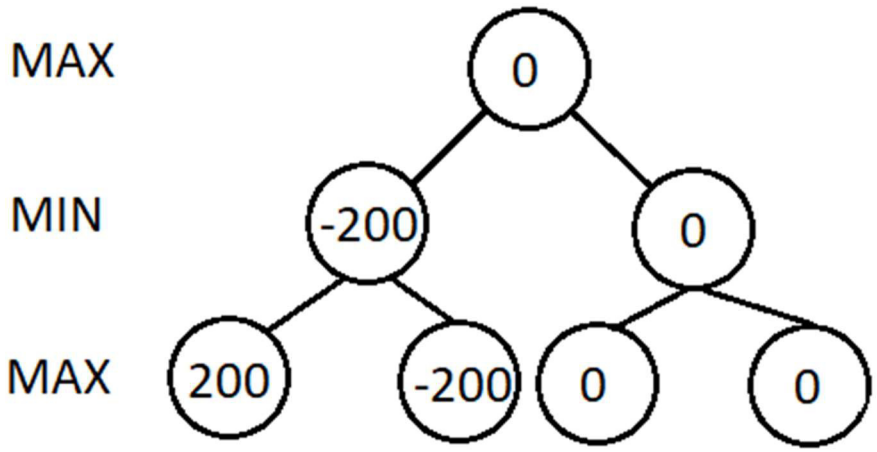

Figure 5: Minimax node selection

We begin by looking at the heuristic evaluation value for each of the leaf nodes (the nodes at the maximum depth). Immediately we can see that the best possible state to end up in results in an evaluation value of 200. However the choice of going to the leaf of 200 or -200 belongs to the minimising player so they will choose -200 as it will put them at an advantage.

On the other side of the tree there are two states with evaluation values of 0. Obviously neither of these states are as good as one with a value of 200, but the minimising player can only make a move which results in a value of 0 here, so it makes more sense for the maximising player to make the move that corresponds to the right side of the tree since it will *minimise their potential loss*.

Alpha-Beta pruning will be included to speed up the bot. While the board evaluation heuristic is performed on a game state the value of that state will be compared against the existing Alpha value (if the move belongs to the maximising player) or the Beta value (if the move belongs to the minimising

player). If the Beta value ever rises above the Alpha value the branch will be pruned as the bot will realise this branch is guaranteed to be worse than one that has already been traversed.

This process will work, but as mentioned in the background section there will be some severe limitations with the Minimax search method. The main limitation is how the game design will lead to an enormous degree of branching which the Minimax search method is not well equipped to deal with.

For example: a *Flusher* unit has a movement range of three. This means, assuming there are no barriers whatsoever surrounding the unit, there are twenty five possible tiles for it to move to. Also, if any of these moves put the unit within attacking range of an enemy unit, another node will be created for the move + attack combination. Being generous and saying that five of these moves will have possible attacks available, this will result in thirty potential moves for a *flusher* unit.

For each of these thirty moves, the opponent will be able to make a similar number of return moves (most likely more potential moves if their selected unit is a *Rusher*, but less if it is a *Shusher*). So each of these thirty moves will spawn thirty more potential moves of their own and if we are simulating to a depth of five moves ahead, this means an approximate 24,300,000 nodes that will need to be simulated.

Simulating all of these nodes will be extremely draining resource wise, even with Alpha-Beta pruning. That's why simulation will need to be fixed to a depth of around five rather than being ten or twenty moves into the future. This is a limitation as a search depth greater than five can be achieved with a different search method.

Mini

Mini will also be a Minimax player, but it will be a much easier version to play against. It will have a much lower fixed depth (for example, two moves) and it will apply a random value to each board heuristic evaluation. The reason for including this player is as a control for when I test Max (Max should be able to comfortably beat Mini) and also as an easier opponent for human players as the other AI opponents will be quite difficult to beat.

Monty

Monty will use the Monte Carlo tree search to decide which move to make next. The tree search method works via repetition and randomness. There are four distinct steps which are repeated until

a fixed duration of time has passed by which time a sizable tree will exist. Like Minimax the tree begins with the root node being the current state of the game. Then the following four steps will begin occurring:

Selection is the first stage. Beginning at the root node, successive child nodes are selected until a leaf node is found. This leaf node is then chosen as the *selected node*. There are many different ways of choosing which child node to select but it is important to choose something that can balance both exploration of new nodes, and exploitation of nodes with high potential for being a good move. I will use **Upper Confidence Bound 1 (UCT)** [9] to achieve this as it balances the weight of under-explored nodes with nodes that have a high win rate.

Expansion is the second stage of MCTS. Unless the leaf node we have selected results in a victory for either player, we will expand it by creating new child nodes and attaching them to it. These new child nodes will represent all the possible moves that can be made from the state of the game stored in the selected node. Now that the selected node has been expanded, we randomly select one of the child nodes attached to it for the next stage.

Simulation is the third stage. After expanding the selected leaf node, we randomly choose one of its new children and simulate game playouts until either player wins. This simulated playout could be completely random (less useful) or it could use some sort of logic so that the simulated playout somewhat mimics actual gameplay (more time consuming). I will attempt both of these methods during my implementation to see if either of them has better results than the other.

Backpropagation is the final step. We take the result of the simulation and update all the way up the branch. No matter the result, the visit counter of each node along the branch will be updated. The win counter for each node will be updated only if the simulation resulted in a victory for that player.

After numerous repeats of these steps Monty will have access to a large tree to help decide which move is best. It will choose a move by selecting the most visited child of the root node: since it will use UCT for the selection of nodes the most visited node will be the node with the highest potential of being a good move.

The MCTS method should be more effective than the Minimax method for playing the game since it will have a higher search depth on promising branches, although it'll be interesting to see just how different the results are when both of these bots compete against one another.

3.7 - Visual Design

In the requirements it was stated that the game needed to be visually accessible. This meant that each unit had to be distinct and the game should display which tiles a unit can be moved to/attack.

The graphics were created using Microsoft Paint. The author of this paper has never been an artist and never will, but the graphics that have been created do their job of being distinct and clear. The units are decent visual representations of how they work; the *Rusher* holds a sword indicating his smaller attack range; the *Flusher* holds a grenade launcher tilted upwards indicating his ability to fire over walls; the *Shusher* is crouched and holding a long sniper rifle indicating his low movement but high attack range.

Figure 6: Green team units

The units are identical in all but their colour for the other team. Different designs for the different teams and different designs if the player was an AI player instead of a human were considered, but in the end it was decided that it being immediately obvious what each unit type is was the most important factor in the visual design and the best way to achieve that was consistency.

Figure 7: Red team units

Similarly, the tile set was designed using Microsoft Paint. The design again is simple which reflects the simple and easy to understand nature of the game board. The design was based on the VR levels of Metal Gear Solid 1 where simplified tiles gave an instant indication as to which parts of the levels were traversable and which weren't without needing to spend too much time drawing actual textures.

Figure 8: Game map tiles

3.8 - Audio Design

Another one of the initial requirements was sufficient feedback for player actions. The easiest way to deliver this feedback is via audio cues. Sound effects will be created to be used for: selecting things in the menu, starting a new game, moving units, skipping a go, firing a weapon, etc. There will also be a jingle for both a human victory and an AI victory which will be played when a match is finished.

To create these sound effects a piece of software called sfxr [10] has been used, an application that has been registered under the MIT free software license. It allows the creation simple sound effects by playing with a number of sliders and these sound effects can easily be chained together to make slightly more complicated ones.

3.9 - Game Structure

From the design sections above it is now possible to think about an overall structure for the game application.

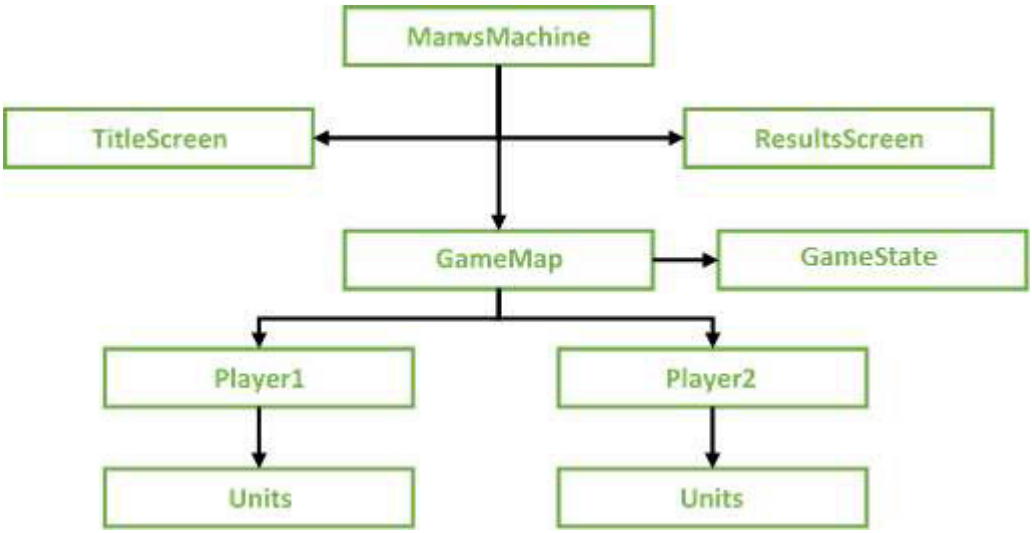

Figure 9: Game structure

ManvsMachine is the main class, it will be responsible for running the entire application. When it first begins it will load and render the **TitleScreen**. The user can make changes to the options on the TitleScreen and the options selected are stored in a GameSetup object. When the user elects to start a match, ManvsMachine will stop rendering TitleScreen and will launch **GameMap** whilst passing it the GameSetup object.

GameMap will use the GameSetup object to configure itself. GameSetup will contain the ID of the map the user has selected and the map will be loaded by libGDX and stored in the GameMap class which can be referenced with ease to, for example, find out if a unit is allowed to move/shoot over certain tiles. The GameMap class will also load the GameState class which contains numerous important methods based on the units stored on the map (such as generating a list of potential moves for a specific unit).

GameMap will also load Player1 and Player2. Depending on the options stored in the GameSetup object these player objects will be either a human player or one of the three AI bots included with the game. To achieve this there will be an abstract class known as **Player** with the subclasses **HumanPlayer**, **Max** and **Monty.**

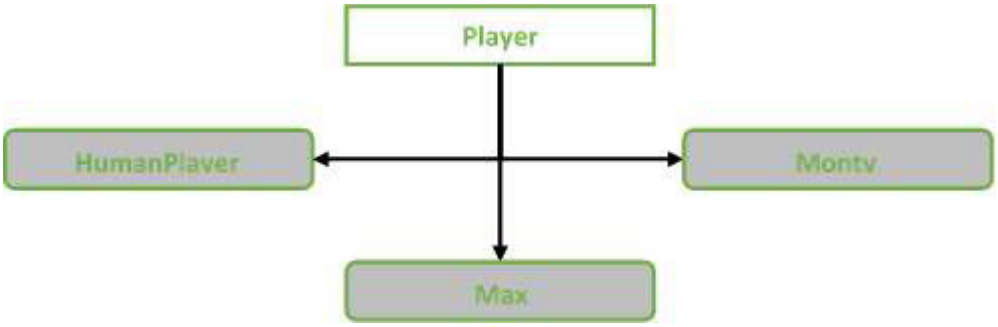

Figure 10: Player abstract class and its subclasses

The **GameMap** class will have a main method which is called repeatedly by ManvsMachine and this main method will ask Player1 to make their move, then Player2, then Player1 and so on until either of the players win. Structuring the player objects in this abstract manner means the main method in GameMap will not have to be cluttered with specific instructions for AI players and human players and can simply call on them to make their move and let the player objects do the work themselves.

Each player will have an array of **Units** which will be loaded as the map is loaded. Each unit will store its own information such as type (which also contains strength, health, movement range, etc), current position, team, current health and more. These units will have a render method which the GameMap class will call after it has rendered the map itself. In this way, the units will be painted onto the screen above the map. Units will also store unique sound effects such as a swoosh to represent the sword of the *Rusher* or a bang to represent the gunshot of a *Shusher*. The units will have methods for moving themselves around the board or shooting other units. Each **Unit** will be a certain UnitType and these UnitTypes will be stored as enums.

When either player loses all of their units, the GameMap class will inform the ManvsMachine class which will then dispose of the GameMap class in its entirety before displaying the appropriate message on the **ResultsScreen** which it will begin rendering. The ResultsScreen will render until a user clicks on it at which point ManvsMachine will begin rendering the TitleScreen class once again.

The specifics of this design will be fleshed out further in the implementation screen but this section should give a good indication of how the overall product is structured and works.

4 - Implementation

The first stage of implementation was creating the project files using libGDX. It provided a framework from which the game code could be written in Eclipse using Java.

4.1 - Game Setup Screen

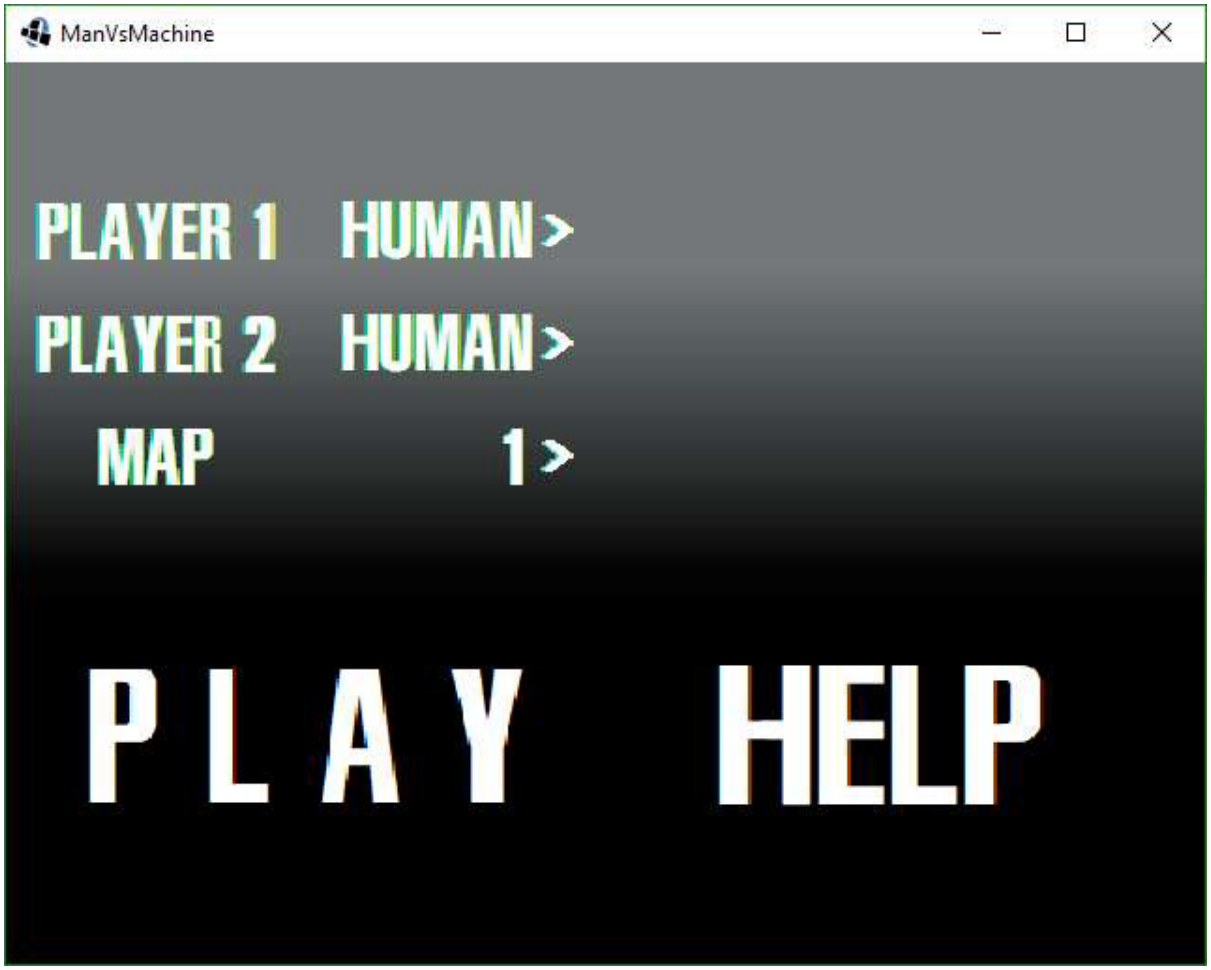

Figure 11: Implemented title screen

The setup screen was fairly simple to implement. It is a collection of picture objects that are rendered above a static background. The update method constantly listens for input and when the mouse is clicked it will check if the click took place within the bounding box of one of the buttons on the screen.

```
63  //          //help button clicked
64              if((clicked.x > 300 && clicked.x < 450) && (clicked.y > 84 && clicked.y < 154)) {
65                  changeSelection.play();
66                  try{
67                      Desktop.getDesktop().open(new File ("ManVsMachineGuide.pdf"));
68                  } catch(IOException ex) {
69                      //issue opening the pdf
70                  }
71              }
```

Figure 12: Code for the help button

If, for example, a user was to click on the 'HELP' button it will open a .pdf file that explains the game rules. The above code works because as soon as the user clicks on the screen the co-ordinates that they clicked are saved as a local Vector2 variable 'clicked'. If the co-ordinates that have been clicked are within the bounding box of the 'HELP' button then the sound effect changeSelection is played and the .pdf file is opened.

Input is handled in this way throughout the code. The libGDX framework takes care of all the details of input handling allowing the developer to focus on what to do with the input triggers rather than worrying about how to collect them.

The current selection of options on the title screen are stored in an object known as a **GameSetup** object. It's an extremely simple object as it stores nothing more than three integers: Player1, Player2 and Map. When the map button is clicked the 'Map' integer in the GameSetup object is incremented by 1 (it uses a modulus function to cap the integer at a maximum value and reset it to 0 when the maximum value is surpassed). Each time a value is changed a function is run which will update the textures on the screen so the changes are reflected visually. The same is true for the Player1 and Player2 buttons and values. When the 'PLAY' button is clicked to begin the game, the GameSetup object is passed up to the ManvsMachine class and then handed down into the newly created GameState class as it helps the GameState class load the correct map and instantiate the right player types.

4.2 - Loading a new game

When the play button is clicked a new game is launched. The first thing that happens is two new player objects are created based on the GameSetup object. Depending on the users selection these new players can be human, Mini, Max or Monty.

Then a new GameMap object will be created with player1, player2 and mapInt being passed in. The mapInt is concatenated with text to form something like 'Map1' and this is then used to access the correct map file from the assets folder. The map file is saved as the type .tmx and the built-in TmxMapLoader in libGDX is used to load the map into the game.

Then the same mapInt is used to load the .json file that corresponds to that map. The .json file contains an array of the units that will be loaded into the map upon launch. The important details are the unit type which must be one of the three unit types included with the game, the x and y co-ordinates and the team.

```
[
    {type:"FLUSHER", x: 9, y: 13, team:0, data:{}},
    {type:"SHUSHER", x: 6, y: 13, team:0, data:{}},
    {type:"RUSHER", x: 9, y: 11, team:0, data:{}},
    {type:"RUSHER", x: 6, y: 11, team:0, data:{}},
    {type:"FLUSHER", x: 6, y: 2, team:1, data:{}},
    {type:"SHUSHER", x: 9, y: 2, team:1, data:{}},
    {type:"RUSHER", x: 6, y: 4, team:1, data:{}},
    {type:"RUSHER", x: 9, y: 4, team:1, data:{}},
]
```

Figure 13: .json file example

There is a specific class in the code to handle loading these .json files. It will read them line by line and use each line to create a new Unit object with the details in the .json acting as the basis for these units. Each of these unit objects are then added to an array until the entire .json file has been read at which point the list of unit objects will be returned to the new GameState class in the GameMap. Each player object is given access to a copy of the list with only their own units present so they can reference their own units with ease.

At this point the GameMap class has access to a complete tiled map as well as a list of units, both of which have been loaded in from the assets folder. The GameMap class will render the tiled map before asking each individual unit to render itself and thus we have a visible game board as well as being able to view all the playing pieces on that board.

4.3 - Game Loop

With everything created and ready to go, we will now continually enter the game loop. Through libGDX the parent class of the game, **ManvsMachine**, will continuously call a method named Update. This method acts as the main game loop which will force the code down certain paths depending on what state the game is currently in. While the game is being played, it will call a method in the GameMap class, also called **update**, which is called over and over again while the game is in progress.

GameMap has a global integer value named currentPlayerInt which will be 0 for player1 or 1 for player2. When the code enters the update method it will first ensure that currentPlayerInt is a valid player number (by running a modulus function on it to ensure it is either 0 or 1) and then it will set a player object called currentPlayer to whichever player matches the currentPlayerInt value. The **update** method will then call the makeMove method on the currentPlayer object. As this piece of code loops repeatedly, it will continually ask the player to make their move. The makeMove method returns a Boolean value which will be true only if the player has completely finished their move.

When this Boolean value is returned as true, the code will enter a new section which will first increment the currentPlayerInt and then check if either player has won the game (and cause the results screen to appear if they have).

```
//check if the currentPlayer has finished their move
boolean moveFinished = this.currentPlayer.makeMove();

//if the move has finished
if (moveFinished) {
    //increment currentPlayerInt
    this.currentPlayerInt++;
    //check if the game has finished
    if (this.checkIfGameFinished()) {
        this.displayResults = true;
    }
}
```

Figure 14: Code to check if a player has finished their turn

Incrementing the currentPlayerInt has the effect of advancing to the next players turn. When the loop is re-entered the currentPlayer object will be updated to store the other player and they will be asked to make their move. The two players will continue to alternate in this fashion until the method checkIfGameFinished returns a true value, indicating that one of the players has lost all of their units.

4.4 - Human Player

When the human player object is asked to make a move it will be asked to call a method based on which phase it is in. As mentioned in the design stage for the gameplay loop there will be two phases during each players turn, a movement phase (Phase 1) and an attack phase (Phase 2). When the user is in Phase 1 it will call into the method phase1.

It is important to note that the player object has a global variable called selectedUnit which holds the unit that they are to use for that turn.

The first step in **phase1** is to check if the current moves available to the player are displayed on the screen. If they are not, the player object will call into the parent object (the GameMap) and call the GameState method to calculate all the moves possible for their selectedUnit. The player now has all of this units' possible moves (every tile they can possible reach) stored as an ArrayList. This ArrayList is passed to the GameMap class so that it can display these tiles on the board.

```
potentialMoves = this.currentMap.getState().calculateMoves(this.selectedUnit);
this.currentMap.displayMoves(potentialMoves);
```

Figure 15: Code to generate and display potential moves

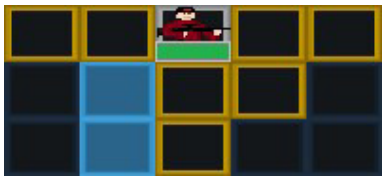

Figure 16: Displayed potential moves

With the game board updated, phase1 now becomes dedicated to listening out for player input. Upon detecting a click the co-ordinates of this click are saved as a Vector2 object. This Vector2 is checked against the ArrayList of potentialMoves and if it matches one of these potential moves then the unit will be moved there. When this happens, the appropriate sound effects will be played, the phase value will be incremented (signalling that phase 1 has finished), the game map will clean itself up (remove the yellow tiles) and similar to how the ArrayList of potentialMoves was acquired earlier, a new ArrayList of potentialShots will be worked out for the unit that has just been moved. These potential shot tiles will also be displayed by the GameMap class.

```
potentialShots = this.currentMap.getState().calculateRange(selectedUnit);
this.currentMap.displayRange(potentialShots);
```

Figure 17: Code to generate and display potential attacks

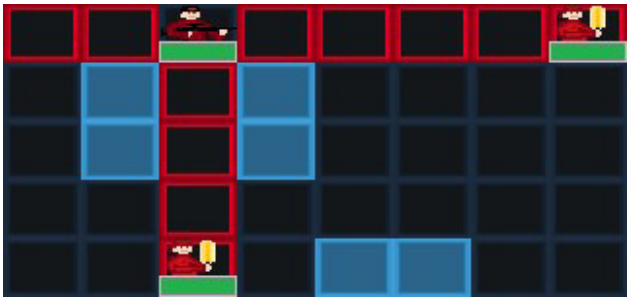

Figure 18: Displayed potential attacks

If the human player clicks on a tile on the board that does not correspond to a tile in the ArrayList of potentialMoves, nothing will happen. If the human player presses the right control key, a method will be called that will skip their entire go and return control to the other player.

Now that the phase value has been incremented, the **phase2** method will be called in place of phase1. This method is just a key listener. It will wait for the player to click on a tile and then it will check if the tile they have clicked on corresponds to a potential shot. If it is a potential shot then the unit will attack that tile. If there is another unit on the tile they have selected, their unit will attack that unit and inflict damage to it based on the strength of their own unit.

If the player clicks on a tile that is not stored in potentialMoves, nothing will happen. The player can also at this stage press right control to skip their go.

The phase2 method will then inform the player that its move has been completed which will cause the makeMove method to enter a final piece of code intended to clean up the entire object and prepare it for its next turn. This involves resetting the global variables and ensuring the phase value is set to phase1. Finally the return value for makeMove is set to true and this true Boolean will signal to the GameMap that the player has finished their turn.

4.5 - Minimax Players

The makeMove method for the Minimax players begins by selecting the player's best possible move. It does this by calling the method miniMaxBestMove. One of the arguments passed into this method is the value *depth* which determines the depth to which the Minimax player will simulate. A higher depth will lead to better move selection so at this point it is possible to limit the strength of the player by allowing this method to be called with a lower depth (this is how the player **Mini** is made weaker than the player **Max** despite them both using the same type of object). Another argument passed into this method is a *copy* of the game state. This allows the player to use this copy to simulate moves without affecting the actual game board they are playing on.

The miniMaxBestMove method is meant to be called recursively. It is easier to view the method itself as a node in the standard Minimax diagram. The first time the method is called we are investigating the root node of the tree (the current state of the game, from which the AI player is going to make their move). It is important to note that this node corresponds to the Maximising player.

The player will then need to calculate every single potential move that they can make from that state. They will do this by calling the method in the GameState class for returning an ArrayList of potential moves, the exact same method used by the human player class. Then the player will physically move the unit to every tile contained in the ArrayList of potential moves and for each spot they will calculate every possible attack using the same method in the GameState as the human player class. In order to streamline this step, every potential attack that does not attack an enemy unit will be ignored (as it will be useless anyway).

```
//fetches and returns a list of every potential move for a given unit in a given GameStateLite
public ArrayList<UnitMoves> getPotentialMoves(UnitLite unit, GameStateLite gameState, int team) {
    ArrayList<UnitMoves> listOfMoves = new ArrayList<UnitMoves>();

    //get a list of all potential movements
    ArrayList<Vector2> potentialMovements = gameState.calculateMoves(unit);
    Vector2 originalCoords = unit.getPos();
    for(Vector2 movement : potentialMovements) {
        listOfMoves.add(new UnitMoves(movement, null));
        unit.moveUnit(movement);
        ArrayList<Vector2> potentialShots = gameState.calculateRange(unit);
        for(Vector2 shot : potentialShots) {
            for(UnitLite target : gameState.units) {
                if(target.getPos().equals(shot)) {
                    listOfMoves.add(new UnitMoves(movement, shot));
                }
            }
        }
    }
    unit.moveUnit(originalCoords);
    Collections.shuffle(listOfMoves);
    return listOfMoves;
}
```

Figure 19: Code to generate an ArrayList of potential moves

The listOfMoves is shuffled before being returned. This is because when I first implemented the Minimax player, at the starting points of the game where the enemy player had no potential attacking moves within the specified search depth they would tend to keep their units in the same place as these moves were the first that were simulated. This would cause the AI player to be incredibly defensive and never attack until the other player had closed in. The returned item, listOfMoves, is of the type UnitMoves which can store both a potential move and a potential attack.

The ArrayList is returned and labelled 'moves'. These 'moves' can be viewed as the child nodes of the parent node in the standard Minimax diagram. The player will then loop through this list of moves and for each one they will create a new copy of the GameState called innerState. On this innerState they will actually execute the move they currently have selected via the for loop. Once the player has executed the moves in the innerState, they will call miniMaxBestMove again.

Although **miniMaxBestMove** is called again the depth value is decreased by 1, the innerState is passed instead of the copy of the parent state and the Boolean value (Max player or Min player) is flipped. The method will continue to be called recursively until a depth of 0 is reached.

In the **miniMaxBestMove** method the first line of code is actually a check for the value of the argument 'depth'. If the value of depth is 0 then, rather than perform any of the previously explained functionality, a board evaluation will be performed on the current state of the game and the value obtained will be immediately returned. This is because the board evaluation should only ever be performed on the leaf nodes.

Each non-leaf node will store a value known as '**bestMoveValue'**. When the first child node of a non-leaf node returns a value (the leaf nodes will return a board evaluation value, the non-leaf nodes will return their bestMoveValue) then the parent of that child node will update their bestMoveValue with the returned value. If the miniMaxBestMove method node corresponds to a Maximising player move, the bestMoveValue will only be updated if one of the subsequent child nodes return a value greater than the current bestMoveValue. Conversely, if the miniMaxBestMove method node corresponds to a Minimising player move, the bestMoveValue will only be updated if one of the subsequent child nodes returns a value lower than the current bestMoveValue.

```
long moveValue = this.miniMaxBestMove(depth - 1, innerState, !isMaxPlayer, (team + 1) % 2, alpha, beta, false);
if(isMaxPlayer) {
    if(moveValue > bestMoveValue) {
        bestMoveValue = moveValue;
        bestMove = move;
    }
    alpha = Math.max(alpha, moveValue);
} else {
    if(moveValue < bestMoveValue) {
        bestMoveValue = moveValue;
        bestMove = move;
    }
    beta = Math.min(beta, moveValue);
}
if (beta <= alpha) {
    //System.out.println("Pruning");
    break;
}
```

Figure 20: Code for updating the current best move in Minimax and Alpha-Beta pruning

Alpha Beta pruning also works here to prune branches as soon as they are discovered to be unviable by breaking the for loop.

This is how the search tree is built in its entirety. Moves are simulated all the way down to a fixed depth (set as 5 moves for *Max* and 2 for *Mini*). These leaf nodes are evaluated and the values are cascaded upwards to inform the top level of the best viable move. Each node will return the optimal value (minimum for the Minimising player, maximum for the Maximising player) and thus the player will be able to, from the root node, select the best possible move by selecting the child of the root node with the maximum value and extracting the UnitMoves object stored within.

Now that the Minimax player has access to the best possible move they can make, the rest of their functionality involves displaying this move in an intuitive way (so that a human player can understand what move their opponent has made). This means it will highlight the board appropriately by changing the tile they will move their unit onto to yellow and then changing the tile they will attack, if any, to red. They will also play the appropriate sound effects.

After this, the player will clean up any global variables and dispose of any simulated game boards etc, before declaring their move over and passing control back to the GameMap class so that it can hand control to the other player.

4.6 - Monte Carlo Player

The overall skeleton of the Monte Carlo player is similar to that of the Minimax player. It begins by selecting the best possible move, then it performs the same functions to display this move visually (sound effects and colouring the tiles) before cleaning itself up and deleting the objects that are no longer necessary. However, the process of selecting a move is completely different.

The method used by the Monte Carlo player to determine its next move is called **findNextMove**. It begins by creating a new **MonteNode** object called 'tree'. The MonteNode object corresponds to a node in the search tree, each one can hold a game state and so they represent a potential move. These MonteNode objects hold a record of their parent node as well as a list of their child nodes. In this way, these MonteNode objects can be strung together to represent a Monte Carlo tree from top to bottom. The object we created called 'tree' represents the root node of the tree and holds the current game state, the one from which the player is trying to decide which move to make.

With this tree created, the player will then acquire a value named endTime by adding 2000 milliseconds to the current system time in milliseconds. It will then enter a while loop which basically states that while the current system time is less than endTime, continue to loop through the following functionality.

```
while(System.currentTimeMillis() < endTime && i < 20000) {
```

Figure 21: Code for Monte Carlo while loop

It is now within this while loop that the main process of MCTS takes place. The first step of this process is **selection** which is where the tree is scanned for the node with the highest potential. The selection begins at the root, it will scan each child MonteNode and select the one with highest UCT value. Then each child of this node will be scanned and the child with the highest UCT value is selected again. This continues until a leaf node (a node with no children) becomes the selected node, at which point that node is returned.

This selection method will result in the node with the highest potential to be returned during each loop. The use of a UCT value helps balance the selection process between branches with high success potential and branches which may not have been explored as thoroughly as the others.

Now that a viable node has been selected, the next stage is to **expand** it. The selected node is passed into a new method which will work out every move possible from that game state and create new MonteNodes to represent each of those moves. These MonteNodes will be added to the child array

of the selected node and the selected node will be added as the parent of each of these children. These potential moves are generated in the same way as they are for the Minimax player.

```
//step 1: selection
System.out.println("Selection");
MonteNode potentialNode = this.selectPotentialNode(tree);

//step 2: expansion
System.out.println("Expansion");
if(potentialNode.getState().checkIfPlayerIsAlive(0) && potentialNode.getState().checkIfPlayerIsAlive(1)) {
    this.expandNode(potentialNode);
}
```

Figure 22: Code for Monte Carlo stages 1 and 2

One of the new children of the selected node will be chosen at random and a playout **simulated** from that node. At this point I had to attempt two different implementations, one where the simulation stage was completely random (a list of potential moves would be generated and one selected at random for each turn) and one where the simulation was guided. The guided simulation worked a lot better as the game has the potential for extremely high branching and completely random playouts did not usually give a good indication of whether a certain state was advantageous or not.

A copy of the game state included in the MonteNode to be simulated is created to be used as the game board for the simulation. The simulation works by having the current player produce a list of potential moves on their turn and run a board evaluation on each one, whichever provides them with the greatest heuristic value is deemed the optimal move. The optimal move will then be executed on the simulated game board.

A maximum of 16 moves will be simulated and if no outright victor has been determined by that point, a board evaluation heuristic is run on the final state to decide which player is 'winning' (whichever player has a higher board evaluation value). The result of the simulation is returned as a SimulationResult object which includes the team integer of the player that faired best from the simulation and a Boolean to indicate whether or not that player *won* the simulation or if they were just in a better position after it.

```
//step 3: simulation
System.out.println("Simulation");
MonteNode nodeForExploring = potentialNode;
if(potentialNode.getChildArray().size() > 0) {
    nodeForExploring = potentialNode.getRandomChild();
}
//System.out.println("Expanding node:");
//nodeForExploring.getState().quickRender();
SimulationResult result = this.simulateRandomPlayer(nodeForExploring);
```

Figure 23: Code for Monte Carlo stage 3

This SimulationResult is used for the final step of MCTS, **backpropagation**. Starting at the MonteNode that has been simulated from the code will add a value of 1 to the visitCount value of the MonteNode. Then it will check which player that MonteNode corresponds to (which player owns the move that the node represents) against which player is stored as the winner in the SimulationResult. If they are the same, the winCount value of the node is incremented. The winCount value will be incremented by 1 if the simulation ended in a victory state, or 0.9 if the simulation ended only with the board being advantageous to one of the players.

Then, the parent node of the currently selected MonteNode will be selected and the same thing as detailed above will happen again. This continues until the results have been propagated all the way up to the root node at the top of the tree.

```
//step 4: back propogation of results
private void backPropogation(MonteNode nodeToExplore, SimulationResult result) {
    MonteNode tempNode = nodeToExplore;
    while (tempNode != null) {
        tempNode.incrementVisits();
        if ((tempNode.getPlayerInt() + 1) % 2 == result.getPlayerInt()) {
            tempNode.incrementWins(result.getGameFinished());
        }
        tempNode = tempNode.getParent();
    }
}
```

Figure 24: Code for Monte Carlo step 4

These steps will continue to be repeated until the two seconds of allocated time have elapsed by which time a large search tree should be present. Selecting the best move from this tree is simple and the player will simply ask the parent node to return the direct child node with the highest visit count. Since the player uses the UCT method of selection in step 1 of the MCTS then the branch that has been visited the most is guaranteed to be the branch with the highest potential of success and thus the optimal move.

4.7 - Board Evaluation

Both the Minimax and Monte Carlo players use a board evaluation function. The board evaluation takes as an input: a state of play. It receives a configuration of units which it can use to produce a value to represent how good or bad that state of the board is for a player.

The first stage of the board evaluation is to check if either of the players are dead. If the current player is dead then a minimum value is returned since dying is obviously the worst possible move for the player to make and thus should be avoided at all costs. If the other player is dead then a maximum value is returned as winning the game is obviously the best possible move for the player to make.

```
if (!board.checkIfPlayerIsAlive((otherTeam))) {
    return Long.MAX_VALUE;
} else if (!board.checkIfPlayerIsAlive(thisTeam)) {
    return Long.MIN_VALUE;
}
```

Figure 25: Code to check if either player is dead in the board evaluation heuristic

The next step of the board evaluation is to check the list of units, each unit is checked one by one. If a unit belongs to the current player and is dead, then a set value is taken away from the board evaluation value. However, if the unit belongs to the current player and is alive, a set value will be multiplied by that units remaining health and added to the board evaluation value.

If the unit belongs to the opposite player and is dead then the set value for that unit is *added* to the board evaluation value. Similarly, if the unit belongs to the opposite player but is still alive, then a set value will be multiplied by that units remaining health and *removed* from the board evaluation value.

```
for (UnitLite unit : board.units) {
    if (unit.getTeam() == this.teamNo) {
        if (unit.isPowered()) {
            retValue += 150;
        }
        if (!unit.isAlive()) {
            switch (unit.getUnitType()) {
            case FLUSHER:   retValue -= (400);
                            break;
            case SHUSHER:   retValue -= (350);
                            break;
            case RUSHER:    retValue -= (200);
                            break;
            }
        } else {
            retValue += (80 * unit.getHealth());
        }
    } else {
        if (unit.isPowered()) {
            retValue -= 150;
        }
        if (!unit.isAlive()) {
            switch (unit.getUnitType()) {
            case FLUSHER:   retValue += (600);
                            break;
            case SHUSHER:   retValue += (500);
                            break;
            case RUSHER:    retValue += (350);
                            break;
            }
        } else {
            retValue -= (100 * unit.getHealth());
        }
    }
}
```

Figure 26: Code for checking which units are alive/dead in the board evaluation heuristic

A final step of the board evaluation occurs only for one of the Minimax players. A random value is added or subtracted to the board evaluation value before it is returned. This random value is only added for the *Mini* player since it is supposed to be an easier challenge than the other bots and thus it will not be as adept at deciding which game state is optimal.

4.8 - Game Units

There is a type of object known as a **Unit**. These objects store all the details for the units in the game such as their current position, their sprite, their team, their current health and many others.

When the render method is called for a unit it will perform a check to see if the unit is still alive by checking the Boolean 'alive' and if it is then it will render the unit sprite at the units current position and the health bar sprite at the same position (but the height of the image is smaller so it only takes up a small portion of the tile). Most of the finer details of rendering the image are handled by libGDX.

```
public void render (SpriteBatch batch) {
    if (this.alive) {
        batch.draw(image, curPos.x, curPos.y, getWidth(), getHeight());
        batch.draw(health, curPos.x, curPos.y, getWidth(), 12);
    }
}
```

Figure 27: Code for rendering a unit and their health bar

When a unit is performing an attack on another unit, the target unit will be passed to the attacker unit's method 'attackUnit'. The attacking unit will play its shoot sound effect and will then call a method, 'shootUnit', on the target which inflicts damage to the target. The argument for this method on the target unit is the strength of the attacking unit.

When a unit is attacked by another unit, the method 'shootUnit' is called on that unit. This method will alter the curHealth variable on the unit by subtracting the strength of the attacking unit which is passed in as an argument. There will then be a check to see if the curHealth value is now less than or equal to 0. If it is the unit will be killed and the appropriate sound effect will play. If they still have health left, a different sound effect will play and their health bar will be updated based on their remaining health.

When a unit is moved, the method moveUnit will be called on that unit. The new position will be passed in as an argument. Since the newPos value will be passed in as tile based co-ordinates rather than pixel based co-ordinates (each tile is 32 pixels wide), these co-ordinates will be multiplied by 32 here so that they represent an exact spot on the game board. Now that the value of the co-ordinates has been updated to represent a position on the board, the curPos of the unit will be changed to the newPos value. Thus when the render method is called again, the unit will now correctly appear in its new position.

```
public void moveUnit (Vector2 newPos) {
    newPos = new Vector2(newPos.x * 32, newPos.y * 32);
    this.curPos = newPos;
}
```

Figure 28: Code for moving a unit to a new position

The different unit types are represented as a UnitType enum. Storing the unit types like this is extremely advantageous as it allows a developer to quickly change details on one line to completely change the stats of a particular unit. The UnitType enum will determine the movement range, attack range, the ability to shoot over walls, the health and the strength of the units. When a new unit is created it will be passed a UnitType and it will pull information out of the UnitType to populate fields such as current health and strength.

```
RUSHER("RUSHER", 6, 1, false, 1, 2),
FLUSHER("FLUSHER", 4, 4, true, 4, 2),
SHUSHER("SHUSHER", 3, 13, false, 2, 4);
```

Figure 29: Code for creating UnitType enums

4.9 - Potential moves and shots

One of the most commonly used methods throughout the entire programme, being used by the human player and both AI players, is the method used to calculate potential moves and potential attack range. These methods are actually fairly simple. **calculateMoves** is a method in the GameState class. The GameState class has access to all of the unit objects and the game map itself and as such is ideally positioned to work out potential moves.

When **calculateMoves** is called, a unit is passed in as an argument and this is the unit whose potential moves we wish to calculate. To work this out, we use a new object type called a **Node** which stores a Vector2 representing a position on the map and an integer called movesRemaining. calculateMoves will create the first Node with the current position of the unit and the movement range of that unit stored in it. This new node is added to a *PriorityQueue* called the **frontier**.

The code will then enter a while loop which will continue to run so long as the frontier is not empty. The first step of this loop is to pull the head node from the frontier, add the node to the **closedList** (a set) and if the value of movesRemaining is greater than 0 then the four tiles surrounding the tile stored in the node will be procured for testing.

There is a method called **checkIfValidMove** which will do the testing. It will check the co-ordinates of a tile against the GameMap to make sure the tile is not a wall. It will also check to make sure there are no other units in this tile. This check is inelegant (check the current position of each unit on the game board against the tile being tested and set a Boolean flag to true if any of the units are in that tile) but effective.

```
//check if a certain cell is a valid move for a unit
public boolean checkIfValidMove(Vector2 coordinatesOfCell) {
    TileType tile = this.map.getTileTypeByCoordinate(0, (int) coordinatesOfCell.x, (int) coordinatesOfCell.y);
    boolean unitPresent = false;
    for (Unit unit : this.units) {
        Vector2 unitPos = unit.getPos();
        if(unitPos.equals(coordinatesOfCell)) {
            unitPresent = true;
        }
    }
    if (!tile.isWall() && unitPresent == false) {
        return true;
    }
    return false;
}
```

Figure 30: Code for checking if a move is valid

If the tile passes the testing and is deemed as a valid move then a new node object will be created with the tile co-ordinates passed through and the movesRemaining from the currently expanded node

will also be added, minus one. This has the effect of diminishing the number of moves remaining as you move further away from the source.

Figure 31: Example of how the movesRemaining value decreases as it moves away from the source tile

The closedList object is a set which prevents duplicate items being added to it. It stores a record of each tile that has been added to the frontier and the frontier is only added to if a move is valid. So once the frontier is empty and every tile within movement range has been checked for validity, the closedList will be a list of every viable move for the selected unit and each element in this closedList is added to an ArrayList of Vector2 objects and returned. Overall this method takes a unit as an argument and returns an ArrayList with the co-ordinates of every tile that unit is allowed to move to from their current position.

The method **calculateRange** is similar to calculateMoves, but different in the sense that a unit is only allowed to shoot in a straight line and not in any direction like when they move. A unit is passed in as an argument and their current position is taken as a source tile.

Next, the code will enter a for loop which runs from 0 up to the attack range of the unit. Each increasing integer is used to check the tile at that distance away from the source block above the unit. Each one of these tiles are passed into a method called **checkRange.**

checkRange will take as an argument, a closedList (similar to the code for potential moves, it is a set) as well as a copy of the unit and the cell co-ordinates. If the tile is a wall tile and the attacking unit cannot attack over walls then the method will instantly return a Boolean value of true. If the tile has a unit on it then the tile will be added to a closedList (meaning it is a viable attack) and if the attacking unit cannot attack over walls then a value of true will be returned. If the tile is just a floor tile with nothing on it then it will be added to the closedList. If the method hasn't returned already it will return a value of false.

```
//check if the cell passed in is a valid shot for the unit
public boolean checkRange(Vector2 cellCoords, Set<Vector2> closedList, Unit unit) {
    TileType tile = this.map.getTileTypeByCoordinate(0, (int) cellCoords.x, (int) cellCoords.y);

    if (tile != null) {
        if (tile.isWall()) {
            if (!unit.overWalls()) {
                return true;
            }
        } else {
            boolean unitOnTile = false;
            for (Unit units : this.units) {
                Vector2 unitPos = units.getPos();
                if(unitPos.equals(cellCoords)) {
                    closedList.add(cellCoords);
                    unitOnTile = true;
                }
            }
            if(unitOnTile && !unit.overWalls()) {
                return true;
            } else if (!unitOnTile) {
                closedList.add(cellCoords);
            }
        }
    }
    return false;
}
```

Figure 32: Code for checking if an attack is valid

The value returned by the checkRange method is saved as a Boolean called ‘blocked’. If the blocked value is set to true, then the for loop will be broken. To explain this simply, if a unit cannot fire over walls then as soon as a wall or target unit is encountered, the for loop is stopped since the attacking unit *cannot* attack over them.

There are three more for loops so that there is a total of four, one for each potential direction that the unit can attack. The tiles stored in the closedList are converted to an ArrayList which is returned by the calculateRange method. Thus this method is similar to the calculateMoves method in that a unit is passed in as an argument and an ArrayList of tiles are returned, these tiles correspond to valid attacks.

4.10 - Balancing Issues

When the game was undergoing testing an issue was encountered. It happened most frequently when a computer player was playing, but it had the potential to occur when any configuration of players were competing with one another.

The issue occurred when both of the players had only one unit left and both of these units only had the health to withstand one hit from the other. In this scenario, an AI player would have no qualms about running away from the other player indefinitely, waiting for them to make a mistake. If two AI players are in this scenario they will continue on in this state almost indefinitely, both avoiding potentially entering a state of certain death. If a human player is playing against an AI player they will likely get fed up waiting and will quit the game or intentionally make a mistake so that they lose.

The solution that was implemented to fix this problem was to allow units to power-up when they take out an enemy unit. Units that are powered up have a shield which allows them to absorb a hit without it affecting their actual health. It also increases their movement range and attack range by one. When a powered up unit is attacked it will lose its shield and the boost in its movement and attack range.

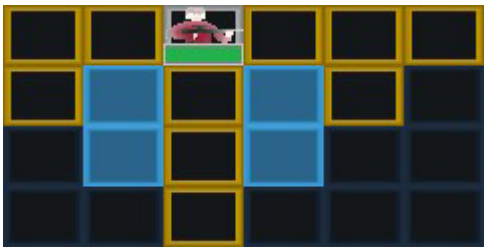

Figure 33: Example of a powered unit having a different texture and increased range

This allows players to gain momentum and the added movement and attack range for shusher and flusher units allow them to pin the speedier rusher units into corners and catch them. The change was successful and minimised the incident of the issue that was being encountered, although it did not completely eradicate it.

The changes were simple to implement. A Boolean value was added to the unit objects named **powered**. The units have a method, attackUnit, which allows them to attack another unit. Now when this occurs the attacking unit will check if the target unit is still alive and if they are not, a new method called **powerUnit** will be called.

powerUnit sets the powered Boolean value to true, updates the texture of the unit and increases the movement and attack range by one.

```
public void powerUnit() {
    this.powered = true;
    this.updateTexture();
    this.setRange(this.getRange() + 1);
    this.setMovement(this.getMovement() + 1);
}
```

Figure 34: Code for powering a unit

There is also a method called **shootUnit** which is called when a unit is attacked by another. Now when this method is called, there will be a check to see if the Boolean value powered is true or not. If powered is true and the attacking unit is not a *shusher* (*shushers* still instantly kill powered units) then a new method called **unpowerUnit** will be called instead of the usual functionality for shootUnit.

unpowerUnit sets the powered Boolean value to false, updates the texture of the unit and decreases the movement and attack range by one (setting it back to the default value).

```
public void unpowerUnit() {
    this.powered = false;
    this.updateTexture();
    this.setRange(this.getRange() - 1);
    this.setMovement(this.getMovement() - 1);
}
```

Figure 35: Code for unpowering a unit

5 - Results

With the development of the game finished it is now important to analyse and study it in an attempt to see if it has met the requirements. The main areas to study and procure results from are the game itself and the two types of AI bot that have been created.

5.1 - Requirements

When the project began a list of requirements were created which would act as a mark of success. These requirements will now be revisited and the extent to which they have been met will be discussed. Some of these requirements were determined in the initial report (and are stored in **Appendix A**), they act as primary requirements:

Create a playable game:

The first primary requirement was to create a playable game. This requirement had numerous sub-requirements.

The game should have a title screen which allows the user to set-up a match:

The game does possess a title screen which allows the user to set-up a match by deciding various starting parameters such as the player types and the map. This requirement *has* been met.

The game should be a 2-player, 2D strategy game where each player controls a team of units and they take turns moving these units around the game board:

This is a good, albeit basic, description of the game loop of the game. It allows two opposing players to take control of a collection of units which they can move around the board. This requirement *has* been met.

Each player will control several, varied units:

There are three distinct unit types and each team contains at least one of each of these unit types. This requirement *has* been met.

The different unit types should have different properties (e.g. weapon range and range of movement):

There are numerous properties that differ the units: attack range, movement range, strength, health and the ability to shoot over walls. This requirement *has* been met.

The winner of the game should be the player who wipes out the opposing players units first:

There is only one win condition for each match. After every move a check is performed to see if either player has lost all of their units. If one of the players has lost all of their units, the game is ended as a victory for the player who still has surviving unit(s). This requirement *has* been met.

The game should be playable by two human players:

It is possible to play a game where the two opposing teams are controlled by human players. This requirement *has* been met.

The game should be playable by two computer players:

It is possible to play a game where the two opposing teams are controlled by computer players. This requirement *has* been met.

The game should be playable by a human player and a computer player:

It is possible to play a game where one team is controlled by a human player and the other by a computer player. This requirement *has* been met.

The game should be visually accessible:

The second primary requirement was to create a visually accessible game. This requirement also had numerous sub-requirements.

Different unit types should be visually distinct:

As detailed in the design section of this report, each unit is visually distinct with careful consideration given to making the unit visually represent how it plays. This requirement *has* been met.

When a unit is selected, its range of movement should be displayed on the game board:

The range of movement for the selected unit is displayed by changing the colour of each tile that corresponds to a viable move to yellow. This requirement *has* been met.

When a unit is ready to fire, its firing range should be displayed on the game board:

The range of attack for the selected unit is displayed by changing the colour of each tile that corresponds to a viable attack to red. This requirement *has* been met.

The game should have a Minimax player:

A Minimax player *has* been implemented, but it has numerous sub-requirements that also had to be met.

This should be a computer player that can play the game in the same way a human player can:

The Minimax player generates its potential moves using the exact same functions as the human player and thus it is limited to playing the game in the exact same way a human player can. This requirement *has* been met.

It should use the Minimax algorithm and a specific heuristic to determine which moves it should make and which it should avoid:

As detailed in the implementation section of this report, the Minimax player does indeed use the Minimax algorithm along with a board evaluation function to determine which moves are more advantageous and which are less so. This requirement *has* been met.

It should employ the use of Alpha-Beta pruning to speed up this process:

As detailed in the implementation section of this report, Alpha-Beta pruning has been employed to limit the number of branches that must be checked in order to decide which move to make. This improved the speed of the Minimax player by quite some amount. This requirement *has* been met.

The Minimax player should not take too long to decide which move it is going to make:

The Minimax player takes a different amount of time for its move depending on numerous factors (the number of tiles the unit can be moved to, the number of units left on the board, etc). On average over the course of a match the Minimax player takes about three seconds for a move but some moves can take up to nine seconds while others are instant. This requirement *has* been met.

The moves made by the Minimax player should make sense:

For a move to 'make sense' it needs to have some form of logic behind it. The player cannot make random moves which put its own units at risk for no reason. From playing numerous games against the Minimax player it can be confirmed that it makes moves that have a sense of logic to them (this is not quite true for the player *Mini* which intentionally makes non-optimal moves at time to make the game easier). Playing against the Minimax player will not be dissimilar to playing against a human player in terms of the logic it shows when making moves, it will attempt to trap opposing units so it can pick them off and it will chase down units that have no chance of putting up a defence. It will not send its own units to their death unless it's due a lack of foresight (only being able to see five moves ahead). This requirement *has* been met.

The Minimax player should have scalable difficulty:

There are two versions of the Minimax player. *Mini* can only see two moves ahead and adds a random value to each board evaluation resulting in the player making non-optimal moves. *Max* can see five moves ahead and does not skew its board evaluations meaning it will present a far tougher challenge than *Mini*. This requirement *has* been met.

The Minimax player should be able to beat a human player:

This will be discussed in more detail later, but the Minimax player has performed very well against human opposition. This requirement *has* been met.

The game should have a Monte Carlo Tree Search player:

A Monte Carlo player *has* been implemented, but it has numerous sub-requirements that also had to be met.

The player is also a computer player that can play the game:

The Monte Carlo player generates its potential moves using the exact same functions as the human player and thus it is limited to playing the game in the exact same way a human player can. This requirement *has* been met.

It should use the Monte Carlo Tree Search algorithm and a specific heuristic to determine which moves it should make:

As detailed in the implementation section of the report, the Monte Carlo Tree Search player does indeed use the Monte Carlo Tree Search algorithm and a board evaluation heuristic to determine which moves are optimal and which are not.

It should not take too long to decide upon a move:

One of the benefits of the Monte Carlo algorithm is that the search process can be halted at any point. This gives the developer the benefit of being able to set the amount of time the Monte Carlo player is allowed to spend on a move. The Monte Carlo player in this game has been given a strict limit of two seconds for a move which means this requirement *has* been met.

The moves it makes should make sense:

As with the Minimax player, the Monte Carlo player always makes moves that are logical and are an attempt to shift the balance of the match in their favour. This requirement *has* been met.

The player should be able to beat a human player:

This will be discussed in more detail later, but the Monte Carlo player has performed very well against human opposition. This requirement *has* been met.

The MCTS player should be able to function without a heuristic so that it will continue to work in different game modes (advanced feature) without additional code:

This was always considered to be an advanced feature and unfortunately it was not met. As things stand the MCTS player is extremely weak when it does not use a heuristic during the simulation stage of MCTS. This requirement *has not* been met.

Secondary Requirements:

There are numerous secondary requirements too. Their inclusion were seen as advantageous but not necessary.

Visual Information:

This requirement mentioned helping the user keep track of whose move it currently is. This has been included in the form of an indicator on the top left of the screen which changes colour to represent the player currently in control of the game. This requirement *has* been met.

Figure 36: The two different team colours that can be displayed on the game screen

Another part of this requirement was for potential moves and potential attacks to be represented visually on the game board. As discussed in the primary requirements section, this requirement *has* been met.

Feedback for player input:

Ensuring that the game provided feedback for player actions was a requirement. This was managed by the inclusion of a wide range of sound effects that are used to signify the click of a button, the movement of a unit, the skipping of a move, a gunshot/sword swipe/explosion, a unit getting wounded and a unit dying. There are also sound effects that play when a match begins and when a match ends (different sound effect for a win and a loss). All of this leads to the game feeling very responsive. If a user was to accidentally click a tile adjacent to a selectable one (when moving a unit, for example) and thus was selecting an invalid move, the lack of any audible feedback would indicate to them that they have not affected the game in any way and thus they would know they have not clicked the right tile. This requirement *has* been met.

Waiting on the AI:

The wait time for the AI players has already been discussed in the primary requirements but this secondary requirement also mentioned the importance of AI moves being broadcast to the player. This is achieved by updating the board when the AI is about to move a unit (the destination tile is changed to yellow) and when they are about to attack (the tile they are about to attack is changed to red). Thus, this requirement *has* been met.

Explanation of game rules:

This was a simple requirement. A game guide has been included which can be launched from the title screen. This requirement *has* been met.

The game should be interesting and fun:

The inclusion of different maps with different configurations of units leads to a variety of different experiences when playing the game. Some of the maps are more open than the others which lead to different playstyles. The variety of units also helps keeps things feeling different as a player with a *flusher* and a *rusher* left will play differently than a player with a *shusher* and a *rusher*. The random nature of the MCTS AI player and the fact the Minimax player shuffles all of its potential moves means that even when multiple games are played between the same configuration of AI players, their moves and strategies will differ.

All of the above lends to the opinion that the game is definitely interesting and the author personally believe the game is fun to play. However, this is a difficult requirement to measure outright and thus an example of future work that should be carried out would be to ask for the opinion of a group of people rather than simply taking the opinion of the creator.

Requirements Conclusion:

Of the numerous requirements laid out for the project, nearly all of them have been achieved. Only one of the primary sub-requirements, to make the Monte Carlo player non-heuristic and thus able to be re-used for different game modes, was not achieved but at the time of writing it was viewed as an advanced feature so this is an acceptable drawback.

Some of the secondary requirements were more subjective and thus it will be up to any future players to decide if these requirements have really been met (if they believe the level of audible feedback is sufficient, for example) but when looked at in a basic manner it is easy to say that they have all been achieved.

5.2 - Game Rules

In order to ensure the game rules are adhered to and the game is correctly functional a series of tests will be performed relating to the different aspects of the gameplay.

Movement:

The movement of units follow strict guidelines. Units cannot move through walls and can only traverse floor tiles. They also cannot move over or through tiles that already contain units. The total number of tiles that a unit can traverse in one turn is limited by their movement range, a value which is determined by their unit type. If a unit is powered up, its movement range will be increased by 1. The movement range of a *rusher* is 5, for a *flusher* it is 3 and for a *shusher* it is 2.

Test	Actual Result
Units may only traverse floor tiles.	The result is as expected. Wall tiles act as barriers for movement and units can only traverse floor tiles.
Units may not traverse tiles that another unit is stationed in.	The result is as expected. Tiles containing other units act as barriers for movement.
Only those moves displayed on the game board as yellow are selectable.	The result is as expected. All of the yellow tiles are selectable (the unit can be moved to any of these tiles) and none of the other tiles are selectable.
Rusher units have a movement range of 5.	The result is as expected.
Flusher units have a movement range of 3.	The result is as expected.
Shusher units have a movement range of 2.	The result is as expected.
Powered *rusher* units have a movement range of 6.	The result is as expected.
Powered *flusher* units have a movement range of 4.	The result is as expected.
Powered *shusher* units have a movement range of 3.	The result is as expected.

Attacking:

The total distance that a unit can attack is limited by their attack range. If a unit is powered up, its attack range will be increased by 1. The attack range of a *rusher* is 1, it cannot attack over walls or other units. The attack range of a *flusher* is 4, it can attack over walls and other units. The *shusher* has no limit to its attack range, it cannot attack over walls or other units.

Test	Actual Result
Only those attacks displayed on the game board as red are selectable.	The result is as expected. All of the red tiles are selectable (the unit can attack any of these tiles) and none of the other tiles are selectable (there is no response from the game).
Rusher units have an attack range of 1.	The result is as expected.
Flusher units have an attack range of 4.	The result is as expected.
Shusher units have no limit to their attack range.	The result is as expected.
Powered *rusher* units have an attack range of 2.	The result is as expected.
Powered *flusher* units have an attack range of 5.	The result is as expected.
Powered *shusher* units have no limit to their attack range.	The result is as expected.
Rusher units cannot attack over obstacles.	The result is as expected.
Flusher units can attack over obstacles.	The result is as expected.
Shusher units cannot attack over obstacles.	The result is as expected.

Units:

The attack and movement range of units have already been tested in previous sections so this section will focus on strength and health. *Rusher* units have a health of 1 and a strength of 1. *Flusher* units have a health of 2 and a strength of 1. *Shusher* units have a health of 1 and a strength of 2. When a unit is powered it will be able to absorb a free hit without taking any damage, unless the attacker is a *Shusher* in which case the attacked unit will still die instantly.

Test	Actual Result
A *rusher* should kill a *rusher* in one hit.	The result is as expected.
A *rusher* should kill a *flusher* in two hits.	The result is as expected.
A *rusher* should kill a *shusher* in one hit.	The result is as expected.
A *flusher* should kill a *rusher* in one hit.	The result is as expected.
A *flusher* should kill a *flusher* in two hits.	The result is as expected.
A *flusher* should kill a *shusher* in one hit.	The result is as expected.
A *shusher* should kill a *rusher* in one hit.	The result is as expected.
A *shusher* should kill a *flusher* in one hit.	The result is as expected.
A *shusher* should kill a *shusher* in one hit.	The result is as expected.
A *rusher* should kill a powered *rusher* in two hits.	The result is as expected.
A *rusher* should kill a powered *flusher* in three hits.	The result is as expected.
A *rusher* should kill a powered *shusher* in two hits.	The result is as expected.
A *flusher* should kill a powered *rusher* in two hits.	The result is as expected.
A *flusher* should kill a powered *flusher* in three hits.	The result is as expected.
A *flusher* should kill a powered *shusher* in two hits.	The result is as expected.
A *shusher* should kill a powered *rusher* in one hit.	The result is as expected.
A *shusher* should kill a powered *flusher* in one hit.	The result is as expected.
A *shusher* should kill a powered *shusher* in one hit.	The result is as expected.

General Functionality:

It is important to ensure that the general progression of the game functions correctly.

Test	Actual Result
When one player finishes their turn, control should switch to the other player.	The result is as expected.
When the health of a unit is reduced to 0, it should be killed and disappear from the board.	The result is as expected.
Whichever map is selected on the title screen will be the one that is launched when the game starts.	The result is as expected.
Whichever player types are selected (human or AI) on the title screen should be the ones that are loaded when the game starts.	The result is as expected.
Clicking the 'Help' button on the title screen should launch the game guide pdf.	The result is as expected.
When a player loses all of their units the game should end and the correct result screen should be displayed.	The result is as expected.
The currently selected unit must increment in a fixed pattern for each player.	The result is as expected.

AI Players:

The AI players are an important aspect of the game and it is essential that they function correctly. They must only make moves which are also viable moves for the player and these moves must be broadcasted to the player properly.

Test	Actual Result
The player *Mini* must only make moves that are viable based on previously established rules.	The result is as expected.
The player *Mini* must highlight move locations with yellow tiles and attack locations with red tiles before making these moves.	The result is as expected.
The player *Max* must only make moves that are viable based on previously established rules.	The result is as expected.
The player *Max* must highlight move locations with yellow tiles and attack locations with red tiles before making these moves.	The result is as expected.
The player *Monty* must only make moves that are viable based on previously established rules.	The result is as expected.
The player *Monty* must highlight move locations with yellow tiles and attack locations with red tiles before making these moves.	The result is as expected.

Graphics:

The interface and graphics of the game must be correct.

Test	Actual Result
The upper left corner of the game screen must indicate which player is making their move by displaying their team colour.	The result is as expected.
All of the units must change appearance when powered up.	The result is as expected.
Players should only have control of units of their colour.	The result is as expected.
The options on the main menu should change when clicked.	The result is as expected.
The correct player colour and team number must be displayed on the results screen.	The result is as expected.

5.3 - Player Strength

One of the main aims of this project was to discover which of the two search algorithms, Minimax or Monte Carlo, provided a more **interesting** and **challenging** experience to a human player. It was stated in the introduction section that the Minimax player was expected to be the strongest since it has the ability to see much further into the future (a higher potential search depth).

This was put to the test by playing numerous matches using different configurations of players. Each type of player played each other numerous times to get an overall picture of which was the strongest. A single human player was used to play each of the AI units as using multiple different players may have led to misleading results since different human players can have different skill levels. The match-ups were *Human* vs *Mini*, *Human* vs *Max*, *Human* vs *Monty*, *Max* vs *Mini*, *Max* vs *Monty* and *Monty* vs *Mini*.

For each match-up, a fixed amount of matches were played with one player type being the home player (player 1) and the other being the away player (player 2) and then the same amount of matches were played with these configurations swapped. The reasoning for this is that certain games have a bias towards players that make the first move (e.g. Tic-Tac-Toe) so giving both players an equal amount of games as the home player was essential to balance the results.

The raw results for this section are stored in **Appendix B**.

Home vs Away bias:

After acquiring the raw data, one of the first things to be checked was if there indeed was a bias towards player 1. This is common in many games, even games such as chess, however it is an important piece of information to know since it may be important for the game in the future. Human players may want to know this as they can then switch positions after every game they play.

From a sample of 300 games, player 1 was victorious in 166 of them (55.33%) while player 2 was victorious in the other 134 (44.66%). Using a z score test for two populations it can be determined that this is a statistically significant difference (with 99% confidence) and thus we can safely say that player 1 has an inherent advantage over player 2 since they can make the first move.

This proves the benefit of, when comparing two players, giving both players an equal amount of matches where they are the player that makes the first move.

AI players as human opponents:

The AI players had to be interesting and challenging. The challenging aspect is easy to measure as the number of wins recorded by an AI player against a human player can be used as a gauge of how challenging they are as opponents.

Mini uses the Minimax algorithm to determine which move it should make next. Mini was specifically made to be a weaker player, it has a shorter search depth (it can only see two moves into the future) and it applies a random value to each of its board evaluations (this can make bad moves seem good and vice versa). A human player and *Mini* played 50 games against one another. The human player won 34 (68%) of these games while *Mini* won the other 16 (32%). From these results it can be determined that *Mini* is not a complete pushover despite its built-in weaknesses and although the human player was much more likely to win any given match, Mini still posed a challenge.

Max also uses the Minimax algorithm but it does not have any built-in weaknesses. It can see five moves into the future and so has a longer search depth and it does not alter the value of its board evaluation. A human player and *Max* played 50 games against one another. The human player won 16 (32%) of these games while *Max* won the other 34 (68%). From these results it can be determined that *Max* is a difficult opponent which requires a degree of skill to beat and thus poses a good challenge.

Monty is the Monte Carlo player included with the game. It is allowed two seconds to decide upon its best possible move and it also has no in-built weaknesses. A human player and *Monty* played 50 games with one another. The human player again won 16 (32%) of these games while *Monty* won the other 34 (68%). These results prove that *Monty* is a difficult opponent and that a human player would need to be skilled to beat it. As with *Max*, *Monty* poses a good challenge.

AI vs AI:

Since both *Max* and *Monty* scored similar win percentages against a human player, it was important to see how they both fare when they face off against one another. *Max* and *Monty* faced each other in 50 games. Of these 50 games, *Max* won 21 (42%) while *Monty* won 29 (58%). We can say these differences are statistically significant (with 90% confidence) which means we can claim that *Monty* is a stronger player than *Max* and it is not purely down to chance that it has won more of the matches.

Max played against *Mini* and out of 50 matches, *Max* won 37 (74%) while *Mini* won 13 (26%). *Monty* also played against *Mini* and out of their 50 matches *Monty* was victorious in 38 (76%) while *Mini* won the other 12 (24%). Both *Max* and *Monty* played extremely well against *Mini* and they share similar

victory percentages so both Minimax and Monte Carlo Tree Search perform similarly against weaker opponents. However as previously stated, the head-to-head record hints that the Monte Carlo player is stronger than the Minimax player.

Posing an interesting experience:

Whether or not the AI players pose an *interesting* experience is a lot more subjective.

For the Minimax algorithm, the fact that there are two difficulties already adds a level of interest since it allows weaker human players to play the game and still achieve results. The skill of the Minimax player can be altered quite easily by increasing or decreasing the search depth (however increasing it above 5 causes it to take an extremely long amount of time to make a move).

It's possible that weaker Monte Carlo players can also be created by giving the algorithm less time to run (for *Monty* it is set at two seconds for searching). Stronger players are also possible, they could be created by giving the algorithm more time to run. In regards to different difficulty levels, both algorithms can be increased/decreased in strength as desired so neither is more interesting than the other in that regard.

Although both *Max* and *Monty* are nearly as strong as one another, it should be said that games with *Monty* are a lot more interesting throughout. The reason for this stems from the strict depth limit imposed on the Max player. Since it can only see 5 moves into the future (including opposition moves) it cannot even see as far forward as the next time it will move the current unit. This problem becomes visible when the AI sometimes moves a unit out into the open only for an opposition sniper to easily pick it off in 6 moves time.

Once the overall unit count has been reduced to 5, *Max* is an almost perfect player since it can now see every single possible move in a single cycle around the game board. Most of its strength comes in the late game but this makes the earlier portions of the match sometimes feel like a bit of a walk in the park as the bot is quite prone to errors.

In comparison, *Monty* can see moves up to an undetermined depth and as such is much less likely to perform weak moves at earlier points in the match when there are 8 units on the board. Although both of the players perform just as well overall, the fact that *Monty* can perform as a strong player from the beginning of a match makes them a much more interesting opponent over the course of an entire match.

This also serves as a potential explanation for the results of *Max* vs *Monty* being so similar and not heavily weighted in favour of *Monty* as predicted at the beginning of the project. It is possible that the apparent advantages of Monte Carlo tree search (spending more time expanding branches that are likely to lead to a beneficial result) would not become obvious unless there is a much larger number of units in play.

Consider a match where each player has 10 units instead of 4, the limited search depth of *Max* would become a lot more of a problem when there are 15 units to be moved that cannot be considered while *Monty* will feasibly be able to visit these potential moves if enough time was given to allow simulation. If *Max* only really becomes a powerful player when there are five units left on the board, it may be crippled by the time this happens in a game that begins with a lot more units. As it is with each team having 4 units, *Max* will always be able to come back in some fashion by the time there are only 5 units left.

5.4 - Results Conclusion

The main aims of the project were to create a playable game and include at least two AI players, one that uses Minimax and one that uses Monte Carlo Tree Search as their search algorithms. Virtually every requirement of the project has been met, the only one that has not been met was always seen as being an advanced feature which there may not be enough time to implement. Basic unit testing was carried out to ensure that the game followed the rules laid out in the design stage and every one of these tests have been passed. From these results it is safe to say that the game itself has been implemented as required and to a satisfactory standard.

The other aim of the project was to discover which of the two search algorithms, Minimax or Monte Carlo Tree Search, was best suited to creating and interesting and challenging AI opponent. Based on the results it would seem that both of these algorithms are adept at providing a good challenge, however the Monte Carlo Tree Search algorithm poses a more interesting and slightly more challenging experience overall. Also, since it can have its move length limited more easily it seems like the best choice for the game moving forward (no player wants to potentially wait 10/20 seconds for the AI to make their move).

Based on these results, the project has been a success. The final deliverable of a game with multiple AI opponents is playable, fun and has followed the design well. The game also acted as a good arena to test the two different search algorithms against one another so that some results could be drawn.

6 - Future Work

There are numerous avenues for potential future work. The game itself could be improved upon by performing further balancing to ensure that no units are overpowered and no tactics exist that make the game too easy. More unit types could be added to make the gameplay more interesting and a larger variety of maps (bigger, different unit configurations, unbalanced starting unit amounts) could be included which would again make the gameplay far more interesting.

The game is also ripe for the addition of advanced features. A map creator would not be unfeasible based on the storage of maps as .json and .tmx files, with more time these features could be extended to allow the creation and saving of these files from within the game. Other game modes such as capture the flag or protect the VIP could be included to extend the gameplay even further.

Another piece of future work would involve the fixing of a rare bug in the game. Sometimes when a Monte Carlo player is making a move it will cause the game to freeze. This issue has appeared so infrequently and so late into the project that insufficient time has been available to discover the cause and fix it. In the same vein, with more time available full and extensive unit testing could be carried out instead of the simple unit testing that has been included in this report.

The idea for making a weaker Monte Carlo player by giving it less time to build a search tree is an avenue that should be fully explored going forward. Creating a Monte Carlo player that does not need a board heuristic to make its move is also an interesting idea that should be considered in the future.

The structure of the code could also do with some polish. For example, creating a single object to manage and store all of the different sound effects may have been a better approach than the one that has been implemented in the game which, even though it works, is a bit messy. There are other areas of code which can be viewed as being untidy which makes the game harder to programme for in the future.

Finally, the game itself could do with visual and audio improvements. The graphics could be updated with more time and the sound effects given a lot more thought. Music could also be included to play during the matches.

Despite the success of the project, there is definitely many ways in which it can be improved and expanded upon.

7 - Conclusion

In conclusion, a playable game has been created which is visually accessible and has Minimax and Monte Carlo Tree Search AI players. The only requirement that wasn't met was the inclusion of different game modes (such as capture the flag) so that the Monte Carlo Tree Search player could be made in such a way that it was re-usable on different game modes without further programming or the creation of different heuristic evaluations.

The main research aim of the project was to determine which of the two search algorithms, Minimax or Monte Carlo Tree Search, was better at posing an interesting and challenging experience. The results have given an indication that the MCTS player is most suited to this, however further testing (on bigger maps with more units) may be necessary to determine this more definitely.

Based on this, the project can be said to have been a success. The game was created to act as a testing ground for the two different AI bots and it succeeded in that regard whilst also acting as a fun game for human players to try out.

8 - Reflection on Learning

It is important after a project of such length and complexity to reflect upon my experiences in order to revisit my expectations and assumptions to see if they have been changed as a result.

One thing that has become abundantly clear to me during my implementation is the importance of a thorough design, right down to the code level. My design covered the overall structure of the system, the visual design, the rules of the game and a bit about how the different AI units were going to work. I believe this design should have gone further and specified more how the various elements within the code were going to communicate with one another.

As an example: when I initially started the implementation of the project, the GameMap class contained the map data, the ArrayList of the units and the player objects. Thus when an AI player wanted to simulate a load of potential game states, it would create copies of the GameMap class. The problem with this is that it had to create copies of numerous bulky objects including the map data instead of simply referring to the original version of the map (the layout of the map will never change during a match). This same problem happened with the unit objects, when copies of the game board were created to represent potential moves, copies of the units were also created including image and sound files.

All of this led to these copied states becoming very load intensive and thus they slowed down the simulation massively, the Minimax player was hit by this and took a large amount of time to simulate even two moves into the future. This was alleviated by re-organising a lot of the objects so that lightweight copies of the GameState and the units could be created which lighted the load considerably. This sort of potential issue could have been caught far earlier if I had specifically *designed* how the AI bots were going to store their simulated states.

So although I knew the importance of design, I had assumed that doing design right the way down to the code level itself was excessive. My assumptions have now changed so I believe that it is a good thing to have a completely thorough design since it allows me to consider and solve potential future issues before they appear rather than as they appear.

Something else that I assumed at the beginning of the project was that for every major development in my system, time was needed for testing. This was an assumption that proved to be overwhelmingly **correct**.

After the stage in which I had implemented the game itself (but no AI players) I gave myself a week of testing. For the most part this involved playing the game repeatedly and in different ways so I could try to break the game in any way that I could. The only changes that came out of this testing were cosmetic (changing the colour of the tile that the unit itself is on from yellow, like the others, to grey) or balancing issues (changing the movement and attack range of units). So this week of time seemed surplus to requirements.

However, when I was finished implementing *both* of the AI types, the week of testing proved to be completely necessary. The Minimax player had numerous issues that were only noticed and smoothed out during the week of testing. For example, since it had a limited search depth it tended to not move its units at all until the opposition player approached. This meant if two AI players were playing against one another, they would both leave their units in their starting positions indefinitely. The Monte Carlo player similarly had issues which were caused by the incorrect set-up of the simulation stage which resulted in skewed results and thus its decision making was severely impaired. This was noticed as the Monte Carlo player was considerably worse than the Minimax players and the issue itself was discovered only via trawling through the code line by line in a debugger. Both of these issues could have been catastrophic for the project if they had not been discovered.

Therefore my assumption that allowing extensive time for testing has *not* changed. Testing ended up being vital to the success of my project and without it two of the most important elements of the project, the AI players, could have been rendered useless.

Another assumption that I had when heading into the implementation stage was the assumption that transferring the basics of the search algorithms (Minimax and Monte Carlo) into practice would be relatively simple. After all, I understood how these search algorithms worked after reading up on them and I understood why they worked the way they did, so surely it would be simple to implement them.

The truth was far from this, many of the examples for each of these search algorithms were on simple games such as Tic-Tac-Toe and so details such as generating a list of every potential move was a lot easier than it would turn out to be for my game. Even the examples that were based around more complicated games, similar to my own, such as chess were not completely transferrable. In chess the moves are made up of movement only. In my game, moves can be made up of moves *and* attacks.

These differences all added up to make the implementation a lot more complicated than I initially assumed. The lesson that I learned here was not to make assumptions based on surface details (the fact I am using basically the same search algorithm as these online examples) and to investigate a bit further. Although I managed to overcome the problems that these assumptions raised, it may have improved the quality of my code if I had allowed myself more time for these portions of the project.

Despite the problems that were caused by some of my assumptions, they have been beneficial for me in a sense. In future projects I will ensure my design is done to a sufficient level and I will ensure that I do not make assumptions based purely on surface level details, I have *learnt* from my mistakes.

On the flipside, the details that I did get right such as including sufficient time for testing have also benefitted me. It is important to realise when you are doing something right so that these correct decisions are reinforced and will continue to be made in the future.

Overall, I have learnt a huge amount from this project. More than just the assumptions that have been changed and reinforced, I have also massively increased my understanding of Java, game design, design in general and testing. I really enjoyed designing and implementing the various AI players and it was quite an unforgettable experience to get beaten by my own AI player for the very first time.

Referenced Material

[1] – http://stanford.edu/~cpiech/cs221/apps/deepBlue.html

[2] – https://www.baeldung.com/java-minimax-algorithm

[3] – https://medium.com/@jonathan_hui/monte-carlo-tree-search-mcts-in-alphago-zero-8a403588276a

[4] – http://mcts.ai/about/

[5] – https://www.gamespot.com/reviews/worms-armageddon-review/1900-2531887/

[6] – https://libgdx.badlogicgames.com/

[7] – http://slick.ninjacave.com/

[8] – http://jmonkeyengine.org/

[9] – https://banditalgs.com/2016/09/18/the-upper-confidence-bound-algorithm/

[10] – http://drpetter.se/project_sfxr.html

Appendices

Appendix A

- create a **playable** game
  - the game should have a title screen which allows the user to set-up a match
  - the game should be a 2-player, 2D strategy game where each player controls a team of units and they take turns moving these units around the game board
  - each player will control several, varied units
  - the different unit types should have different properties (e.g. weapon range and range of movement)
  - the winner of the game should be the player who wipes out the opposing players units first
  - the game should be playable by two human players
  - the game should be playable by two computer players
  - the game should be playable by a human player and a computer player
- the game should be **visually accessible**
  - different unit types should be visually distinct
  - when a unit is selected, its range of movement should be displayed on the game board
  - when a unit is ready to fire, its firing range should be displayed on the game board
- the game should have a **Minimax player**
  - this should be a computer player that can play the game in the same way a human player can
  - it should use the Minimax algorithm and a specific heuristic to determine which moves it should make and which it should avoid
  - it should employ the use of Alpha-Beta pruning to speed up this process
  - the Minimax player should not take too long to decide which move it is going to make
  - the moves made by the Minimax player should make sense
  - the Minimax player should have scalable difficulty
  - the Minimax player should be able to beat a human player
- the game should have a **Monte Carlo Tree Search player**
  - this player is also a computer player that can play the game
  - it should use the Monte Carlo Tree Search algorithm and a specific heuristic to determine which moves it should make
  - it should not take too long to decide upon a move
  - the moves it makes should make sense
  - the player should be able to beat a human player
  - the MCTS player should be able to function without a heuristic so that it will continue to work in different game modes (advanced feature) without additional code

Appendix B

Player 1	Player 1 Wins	Player 2 Wins	Player 2
Human	18	7	Mini
Human	11	14	Max
Human	9	16	Monty
Mini	9	16	Human
Mini	7	18	Max
Mini	7	18	Monty
Max	20	5	Human
Max	19	6	Mini
Max	12	13	Monty
Monty	18	7	Human
Monty	20	5	Mini
Monty	16	9	Max
Total	166	134	Total