# Conquerors of Catan

Building an AI player for The Settlers of Catan

Supervisor: Professor John Gan

Second Assessor: Doctor Ana Matran-Fernandez

Course: Integrated Master in Computer Science

Harrison Phillingham - CE2001418

April 28, 2023

## Contents

## Abstract

The Settlers of Catan (Settlers) is a classic board game that provides a unique and interesting challenge to modern AI methods, such as those found in AlphaGo. In Settlers, players build and develop roads, settlements and cities on a board of resource tiles, scoring them Victory Points of which they need 10 to win. However, the challenge of making an AI player for this comes from multiple factors, including imperfect information, elements of chance and negotiation, and having more than 2 players. In this project, I aim to first produce a fully-playable command line version of the game, and then create a strong AI player with multiple strategies that can be investigated and compared, aiming for human-competitive level or better.

## Acknowledgements

I would like to express my gratitude to my family; my mother Tracey and my sister Esther, and especially my father Gaius. Without their exceptional skills in the game (resulting in my constant losing), I would not have had the motivation or desire to create an AI player. I would also like to thank them for proofreading this report, providing useful corrections and suggestions from an outside perspective, and for celebrating in the joys when a new feature of the project was complete.

Secondly, I would like to show my gratitude to my supervisors, Professor John Gan and Doctor Ana Matran-Fernandez, for suggesting ideas and being a sounding board for potential developments.

Finally, I would like to dedicate this report to Klaus Teuber, the creator of Settlers of Catan, who sadly passed away on April 1st, 2023 during the development of this project. His creative and strategically challenging game has stood the test of time, and is a classic found on most enthusiasts' shelves. May it long continue to be this way.

## Symbol List

**Table 1:** List of Terms and Meanings that appear throughout the report

| *Term* | *Reference* |
|---|---|
| AB Pruning | Shorthand for Alpha-Beta Pruning, a method used to speedup a MiniMax algorithm (see Section 2.2.3). |
| A Move's Type | The main part of a move, e.g. Placing a settlement, Placing a road. |
| A Move's Options | The options for a move, such as the location to place a settlement. |
| A Move + Options Combination | The combination of a move and its option, e.g. placing a settlement at `a1`. |
| Board (Settlers) | The main board in Settlers, made up of 19 resource generating tiles. |
| City (Settlers) | An upgraded version of a settlement. Generates 2 resources when the dice is rolled for the tiles it is on. |
| Development Card (Settlers) | Can be bought, and is similar to a chance card in Monopoly. |
| Dice Roll | 2 dice that are rolled to generate resources for the tile(s) that match the sum of the roll. |
| Heuristic / Heuristic Function | A method used to consider how good a game position is. |
| Game Tree | "A directed graph whose nodes are positions in a game (e.g., the arrangement of the pieces in a board game) and whose edges are moves (e.g., to move pieces from one position on a board to another)" [1] |
| MiniMax | An algorithm designed to traverse a game tree, and return the best potential move by minimising and maximising alternate player's turns. |
| Node | A position in a game tree |
| Match | A single game played. Often (but not exclusively) used when referring to one game in a set of games to be played. |
| Move | An action a player can make. Connects two nodes in a game tree. |
| Port (Settlers) | Used to make better value trades if a player has built a settlement on it. Can either be 3 for 1 wild, or 2 for 1 for a specific resource. |

| *Term* | *Reference* |
|---|---|
| Program | The program that a potential user would run, created as part of the project. |
| Project | The overall project that the author has worked on. |
| Resource (Settlers) | The five main resources in the game; Rock, Clay, Wheat, Sheep and Wood. |
| Resource Card (Settlers) | Generate by the dice roll and tiles, and used to build roads, cities and settlements, and buy development cards. Also used to trade with. |
| Resource Frequency (Settlers) | AKA Tile Frequency. How likely a tile and resource are to be rolled, based on it's dice roll. For example, 12 is much less likely than 6 or 8. See Table 7. |
| Road (Settlers) | Used to expand a player's reach and allow them to build settlements in more places. |
| Robber (Settlers) | Game piece moved when a 7 is rolled. Stops production of a tile, and robs a player who is on that tile. |
| Roll Map | A statistic containing the distribution of a player's settlements and cities, per resource and resource frequency. Used to evaluate which resources a player is more likely to get. See Section 6.2.1. |
| Settlement (Settlers) | Basic building that generate 1 resource when the dice is rolled for the tiles it is on. Must be placed two roads away from any other settlements. |
| Strategy (Settlers) | Any plan of action for the game. Examples include aiming for more Rock and Wheat for cities, or more Clay and Wood for roads. |
| Sub-tree | Any smaller part of the main Game Tree, made of a node and all it's children. |
| Tile (Settlers) | A heaxgon shaped part of the board, which may generate resource cards for players built on it. |
| User | A human who interacts with the program. |
| UX | Shorthand for User Experience. |
| VP(s) (Settlers) | Shorthand for Victory Point(s). |

# 1 Introduction

## 1.1 Project Objectives

The Settlers of Catan is a game that the author's family love to play, however it is a game that the author rarely wins. Whether it's being beaten at the last minute, or never really progressing past a half-dozen Victory Points, a victory never seems to materialise. Therefore, the author has decided to try and create an AI player for this game to hopefully be able to finally get conquer the island, and be a true Settler of Catan.

There are two main project objectives. The first is to create a fully functioning command-line version of The Settlers of Catan, which will be done in Python, and the user(s) will interact via the keyboard. The recreation should be as close to the original as possible and be familiar to anyone who has played Settlers before.

The second objective is to develop an AI that will play this digital version of Settlers, and to formulate multiple strategies for the AI. Then, these will be tested to test find the best one. Finally, when the best strategy has been found, the AI should play against a human to see how well it performs. The AI Player should also work reasonably quickly (the original minimum benchmark identified for this is an average turn time of 15 seconds, with the ideal average time being 5 seconds), so that a human player does not lose interest.

## 1.2 Report Overview

In this report, the author will first give a brief explanation of how to play Settlers of Catan along with the algorithm that will be used for the AI. They will provide an insight into the work that has already been conducted into AI players for it. They will explain their implementation of the game, including how they have created the board, the user interaction, and how they have designed a basic random player to play against. After this, the author will explain the more advanced AI player. This includes the basic starting strategy used to guide its moves, methods used to speed up performance, and the more advanced strategies later discovered and implemented and their performance. The author will then touch on their functional and UX testing methods, before demonstrating the performance of this more advanced AI. This will involve comparisons against both the basic random AI, and also human players. Finally, they will summarise their work, including the project management process, technical achievements and any future works they wish to add.

# 2 Background

## 2.1 A Brief Introduction to The Settlers of Catan

The Settlers of Catan (Settlers) is an award-winning board game created in 1995 by Klaus Teuber set on a fictional island, where the aim is to try and be the first player to construct a large enough civilisation. This is defined by a player reaching 10 Victory Points, and players achieve this by collecting resources, trading with others players, and building roads, settlements and cities using these accumulated resources. Each settlement is worth 1 Victory Point, each city is worth 2 Victory Points, and the longest road and largest army cards (given for either having the longest road, or having played the most Soldier development cards) are worth 2 Victory Points each. Players may also buy a development card that is simply a single Victory Point.

The board contains a grid of hexagon shaped tiles, each of which may produce a specific resource for those built on them. Each tile has a number on it indicating the dice roll needed for it to produce a resource, as well as a series of dots showing how likely that roll is to happen. The numbers in red are the numbers statistically most likely to be rolled. In the example below in Fig. 1, when an 11 is rolled, the red and blue players both receive a single wheat card for the upper-left-most wheat tile, but the lowermost wheat tile doesn't produce anything as no-one has built on it.

The game is played over multiple rounds, in which each player gets a turn. During their turn, a player first rolls the dice to generate resources. They then may choose to trade with either another player, trade with the bank in a 4:1 deal (giving up four of the same resource for the choice of a single other). They could also trade with a port if they have built on one, which are either 3:1 wild, or 2:1 for a specific resource. Finally, a player may choose to build either a road or a settlement, upgrade a settlement to a city, or buy a development card. These are similar to chance cards in Monopoly and can only be played the turn after they are received.

If a player rolls a 7, then the robber token is moved to a tile, which stops that tile producing when it is rolled until the robber is moved again. The player who rolled the 7 then steals a random card from a player who has built around that tile.

A full version of the rules can be found on the Catan website [3].

## 2.2 Technical Background

### 2.2.1 MiniMax

The author has chosen to utilise the MiniMax algorithm [4] as the basis of the smart AI player. This algorithm searches a tree of moves [1], and evaluates positions, aiming to minimise the losses of the player, and maximise the gains where it can.

It starts by forming a search tree of all possible moves and resulting states. The levels of the tree alternate who's turn it is. An example of this can be seen in Fig. 2, where 'min' represents the opponents turn and moves, and 'max' represents the MiniMax players turn and moves. The current game position is the node at

**Figure 1:** An example Settlers Board [2] showing the board, three different players, the robber and the resource and development card piles

the very top and the resulting position of each possible move is placed on the row below. Then, each position that can be reached from those positions is placed below, and so on, up until a pre-determined limit. After this construction phase, the algorithm moves down the first branch of the tree, assigning each node a score or ranking from a *Heuristic Function* (explained more in Section 2.2.2) to symbolise how good that position is. A higher score represents a better position for the algorith, Once this is complete, the algorithm starts at the bottom nodes of the current branch (known as the leaf nodes) and compares all the nodes that are the children of the same parent node. Then, it will either return the minimum value of all the nodes if it is the opponents turn (because the player assumes the opposing player will do the most hurtful move to it), or the maximum value of all the node (to maximise the gains from our turn). The value is returned to the node above it. This happens repeatedly for all branches of the tree until all the immediate nodes that are directly connected to the top node have a score. Finally, the algorithm will pick the move with the highest score to perform. This process can be seen again below in Fig. 2, where the red arrows represent the move returned to the above node and the overall chosen move at the top being shown with the blue arrow.

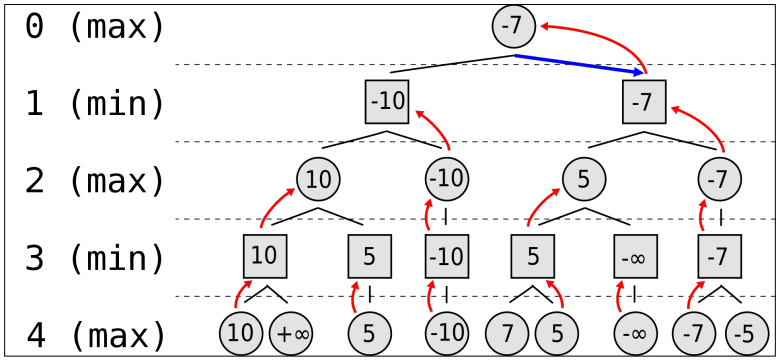


**Figure 2:** Example MiniMax Tree [5]. Circles and Squares represent game positions, and black lines represent moves between them.

Programmatically, this function is typically implemented recursively, where it calls itself at each position for each child, and that instance will call itself. This is done until it reaches the bottom, where it will evaluate the position, and return the score to the function that called it. A more detailed explanation of the author's implementation can be found in Section 6. The author chose to use this method as it allows for a lot of research and investigation into which strategies work the best. Other methods, such as Monte-Carlo Tree Search [6], do have their benefits, but do not have a user defined heuristic/strategy, and so don't allow for as much investigation or research.

#### 2.2.2 Heuristic Functions for Evaluating Game Positions

Heuristic functions [7] are methods used to evaluate a game position by factoring in statistics such as the current player's positions and return a score for that position. Heuristic Functions are used in many games and their main purpose is to get a good-enough evaluation of a position quickly. They can be used in a game as simple as tic-ac-toe, to as complex a game as chess. Therefore, heuristics are more focused on speed of

evaluation than optimality or completeness of the evaluation. A basic implementation of a heuristic function for Settlers could simply be returning the number of victory points that the player has. This is obviously not a complete score, as it does not factor in things such as how many cards a player has, whether they could be just about to gain a point, or how other players are doing, but it is a rough estimate of how well a player is doing. This is also the one that a human would most often use, as it is visible on the board at all times.

MiniMax, like all heuristic search algorithms, uses a heuristic function to evaluate game positions in order to understand how a move will affect the current player. They form the backbone of the search tree, as without them, MiniMax has no idea how good a position is, and the quality of the heuristic function greatly affects how well the MiniMax algorithm performs. At leaf nodes of a game tree where a game has ended, heuristics typically returns an extremely large number for wins or an extremely low number for losses, such as a scores of $\infty$ and $-\infty$ respectively. This is done as there will be no better or worse scores than these, and also so that these scores are always propagated up the tree.

#### 2.2.3 Alpha–Beta Pruning

Alpha-Beta (AB) Pruning [8] is a method used to decrease the number of nodes that need to be evaluated by the MiniMax algorithm. Due to the nature of this algorithm, it is possible to stop evaluating any further nodes of a subtree if a node has already been found that is worse than a previously examined move. In the example of Fig. 3, the entire right subtree of the node of 5 on the second-highest row can be ignored. This is because this row is looking for the minimum value. As 5 is smaller than 8, anything in the subtree of 8 will not be counted, and so needs not be evaluated. [9] states that "For a chess playing program, searching to a depth of 4-ply with an average of 36 branches from each node, there are more than 1 million terminal positions. Under optimal conditions, the Alpha-Beta algorithm would reduce the number of terminal nodes examined to around 2000, or a saving of 99.8%". Alpha-Beta pruning also benefits greatly if the nodes are ordered correctly, as fewer comparisons need be made if a good first value is found. These combined show the power that AB can have in increasing the efficiency of the MiniMax algorithm.

The author uses AB pruning to speed up the MiniMax search tree, found in Section 6.3.1.

#### 2.2.4 Potential Issues with MiniMax

The use of MiniMax in a game like Settlers is not straight forward. The MiniMax algorithm is typically used in games where the following conditions are true:

- There are only 2 players. As a result, the author will need to extend the MiniMax algorithm to compensate for the additional players. This will need to be done in a way that works for any number of players between 2 and 4 (as per Settlers rule), or potentially even greater.
- All players know all information (perfect information). Settlers contain hidden information, such as resource cards and development cards. Therefore, the author will need to identify how best to navigate this issue and ensure the MiniMax player is able to work well if it does not know all the information.

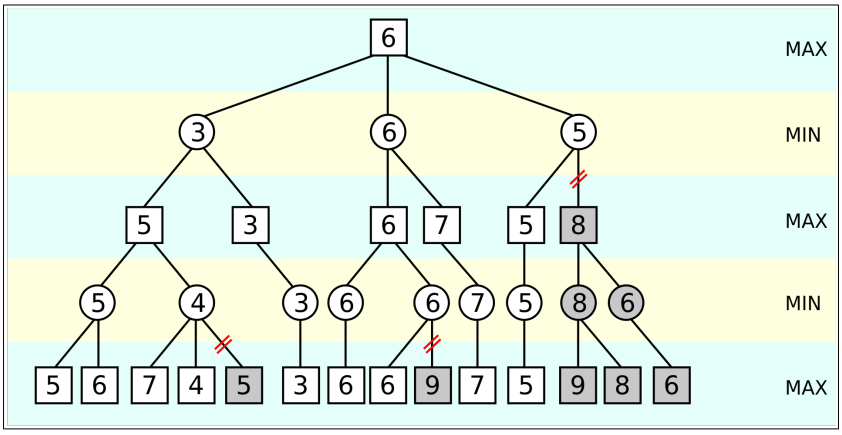


**Figure 3:** Example of Alpha-Beta Pruning [10]

- There is no random chance (deterministic). The resources that players receive are based on the dice roll, and so the MiniMax algorithm will need to factor this in. Closely linked to this is the idea that MiniMax is not often used in games where moves have a cost. The author will need to code the algorithm in such a way that this is not an issue, as otherwise the algorithm's ability to look ahead will be limited.

### 2.3 Domain Background and Literature Review Summary

Due to the above-mentioned issues, Settlers is not an obvious choice as an application for game tree algorithms. However, the author was able to identify the following information from a selection of papers.

One method used to help avoid the issues relating to imperfect information is the Szita rule simplification, discussed in [11]. Szita et al. suggest that the elements of imperfect information (which cards are in players hands, which cards are stolen using the robber) can be made public, as this information often becomes apparent quickly anyway. As well as this, they remove any trades, except from trading with the bank. These modifications are done to enable the MCTS AI to perform the search easier and quicker. Later on in the paper, Szita et al. discovered that their MCTS player performed well while using some well-known Settlers strategies such as rock and wheat.

Roelof [12] discusses the use of 'Expectiminimax' [13] in combating the random chance element of Settlers. The idea behind Expectiminimax is to add chance nodes to the search tree, so that the weight of a chance node is the probability that it's child is reached. For example, a chance node for gaining a card from a frequent rolling tile would have a higher weight than one for a less frequency rolling tile. This allows for random chance to be accounted for during the searching of the tree.

Austin et al. in [14] suggestion some initial placement strategies to help get a better start to the game. The paper is designed as a proof that board games can be used to teach probability in the classroom, yet the suggestions they make are interesting and could be applied to the program. A selection of the concepts behind the strategies they describe [14, pp. 278–281] are included below:

- Possibility for Expansion
  - One key element that Austin et al. writes about are 'Neighbours' strategies, or strategies for placement based on the nearby tiles that the player is likely to expand to later in the game. Instead of focusing on the single location, this strategy takes into account future developments, and how well the player is setting themselves up for later in the game. The author's do make an interesting addition, that positions become less valuable if more people build there, as the resources are no longer rare, and expansion is limited.
- Resource Rarity
  - A second concept they provide is resource rarity. This is explored in the context of the frequency that the tile numbers are rolled, and therefore that more cards are gained, but also on which resources are actually beneficial. For example, clay and rock are more rare than other resources, and therefore are slightly more valuable. However, wheat is a key resource in building both settlements and cities, and so this is also a key resource.
- Building Requirements Strategy
  - Similar to the last point, this final concept focuses on gaining specific resource combinations for constructing buildings. For example, a player building on rock and wheat will much more likely be able to build cities in the game, and so this provides a natural placement pairing for the first two settlements.

Guhe et al. in [15] also provide some interesting strategy ideas. For example, [15, Figs. 3,4,5] shows that the number of wins a player receives increase as the number of settlements and cities increases (as is expected), but when development cards are favoured, the win rate lowers. This is in contrast to some Settlers players who strongly favour development cards due to their random nature. In their research, Guhe et al. also found that favouring the longest road, and disfavouring the largest army aided their player's win rate. The author suggests that this could be related to the fact that the largest army has to come from development cards, which was previously shown to be not as beneficial, whereas the longest road comes from building structures, and also aids in expanding to new areas.

# 3 Implementation Overview

## 3.1 Tools Used and Intellectual Property

To create the project, the author used Python [16], along with JetBrain's Pycharm Professional IDE [17]. GitLab was used as the remote code host and Atlassian's SourceTree [18] and JIRA [19] programs were used to interact with the remote repository and keep track of tasks respectively. The pre-commit framework [20] was used to perform formatting and styling tasks, such as applying the code formatting style 'black' [21] to the code. Some basic shell scripts were also created to help with menial tasks, such as generating the dependency graph shown below, or counting the total number of lines in the program. These can be found in `/src/dev/`.

Aside from a section in `longest_road.py` written by the AI Language Model ChatGPT, all source code was written by the author. The only external assets that have been used are the emojis, which are from the Apple Color Emoji Font [22].

## 3.2 Code Structure

The source code for the project is contained within the `src/` directory in GitLab and is broken up into multiple files to help with organisation. In Figure 3, you can see the dependency graph created by the pydeps module [23], which analyses the byte code in order to find imports. Table 2 shows a list of files in the `src/` directory, and what code they contain, to give a broad understanding of the project.

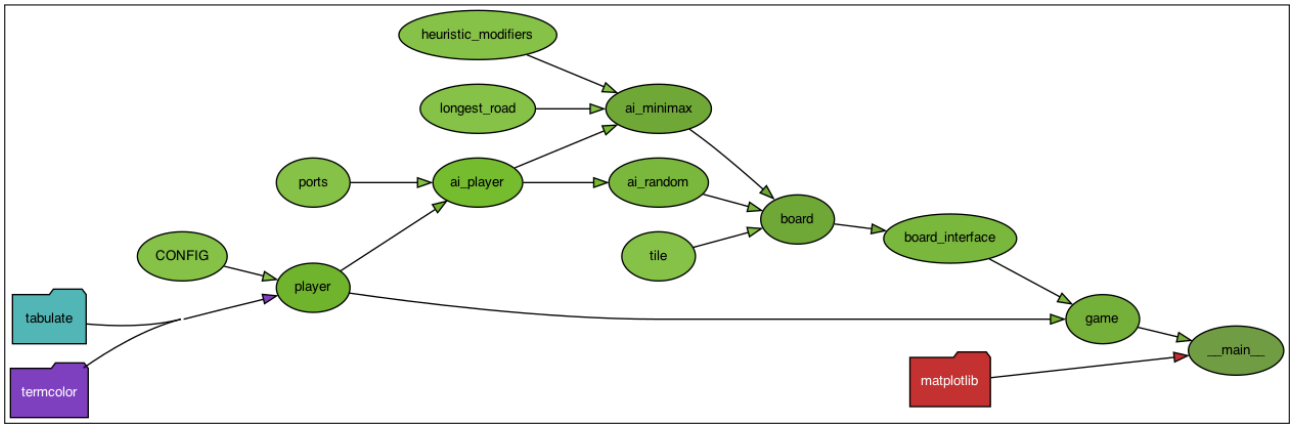


**Figure 4:** Dependencies Graph, created using the `pydeps` module, 29/3/23

**Table 2:** Files in `src/` directory, in reverse import order

| *File* | *Use* |
|---|---|
| `CONFIG` | File containing configuration variables for the program, stored in a python `dict`. |
| `player` | The default player class. Includes all methods a player needs to choose a move. Implementations are included for a human player to use. |
| `ports` | Contains port functions used within the `ai_random` and `ai_minimax` classes. |
| `ai_player` | The template class for all other AI players. Overwrites the human interaction methods with blank ones that are then implemented in the subclasses. |
| `ai_random` | Random player class, that extends the `ai_player` and provides method implementations for actions that are chosen at random. |
| `heuristic_modifiers` | Contains the code that modifies the default `ai_minimax` heuristic to create different strategies. |
| `longest_road` | Contains the code for the methods to calculate and find the longest road. The file is imported here as `ai_minimax` needs this for it's heuristics. |
| `ai_minimax` | MiniMax player class, again extending the `ai_player` class and overriding the methods with it's own MiniMax versions. |
| `tile` | Simple tile class that contains basic information about the 19 board tiles, including their resource, number and symbol. |
| `board` | Board class, which contains all of the tile and card deck data, as well as building locations and other game information. Is used to display the board to the user. |
| `board_interface` | Interface class, which provides a standardised way for all players to interact with the board using functions such as `place_settlement()` or `rob_player()`. Also gives and receives the cards when the dice is rolled, and makes sure the game is in a correct state. |
| `game` | Game class, providing basic methods to setup, run, and keep stats of a game. Was created to allow multiple games to be played sequentially by creating multiple game class instances. |

| *File* | *Use* |
|---|---|
| `__main__` | Main entry point for the program. Creates matches using the game class, and provides a menu to configure the game options before running. Also provides stats and graphs at the end if multiple matches are played. |

`termcolor`, `matplotlib` and `tabulate` are all external libraries not created by the author. `termcolor` is used to colour the terminal text, and `matplotlib` and `tabulate` are used to create graphs, and command-line tables respectively.

### 3.3 Interaction of Key Classes

To give a brief overview into the interaction between these classes, the author has drawn Fig. 5 representing which classes talk to which during the program.

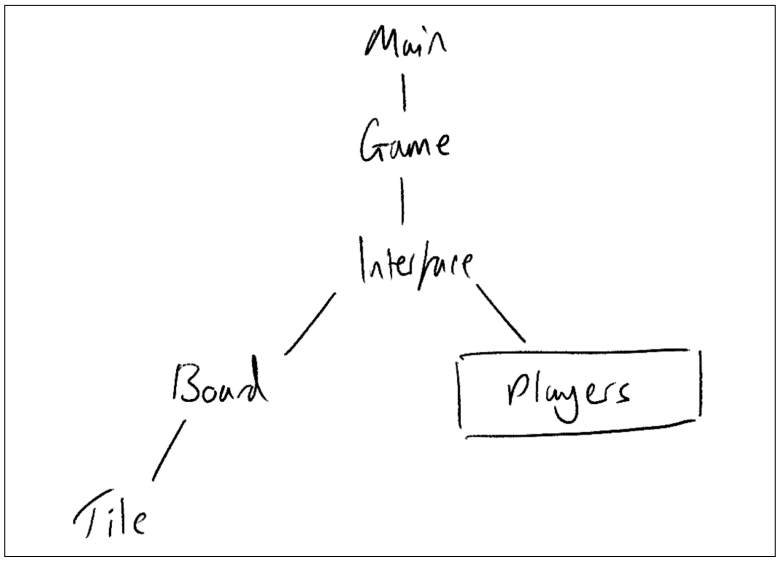


**Figure 5:** Hand-drawn Interaction Diagram between key classes

- Main File: The main file talks to the game object in order to start the game, or to collect stats at the end of the game(s).
- Game Class: The game class talks to the interface in order to tell it the dice roll, or to get players to place settlements during setup, or perform actions on their turn. It does not talk directly to players or the board.
- Player Classes: The player class tells the interface which move it wants to make.
- Interface Class: The interface takes the move the player wants and performs it, and also gives players their cards or takes them away when played.

- Board Class: The board class is modified by the interface, and does not talk to the other classes.
- Tile Class: 19 instances of the tile class make up the board in the board class. They only hold data, and are not talked to by other classes.

### 3.4 Other Notes

The program is designed to serve two purposes. The first is research, in which the game is configured and then played without human interaction; the only players are AIs. Multiple matches are normally observed in this mode, and the games are played without the focus on smaller tactics, but on larger strategies. The second purpose of the program is playing with a human, which puts the research into practice by creating a good AI. This mode typically only includes 1 match, and focuses on the human interaction, and therefore has longer pauses for the human to observe the AIs behaviour.

# 4 Digital representation of Settlers

## 4.1 Board Layout and Display

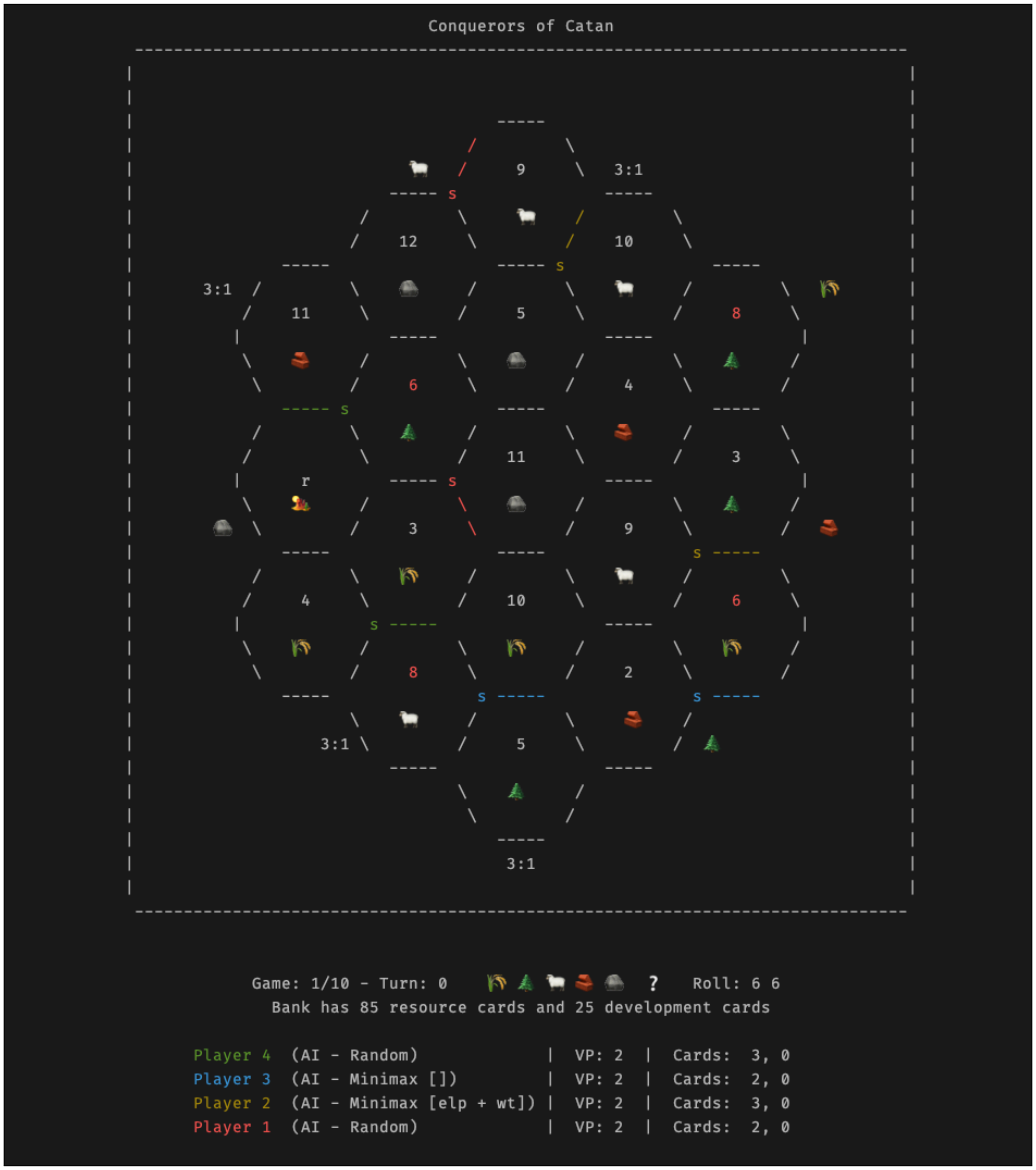


**Figure 6:** An example board

Fig. 6 shows an example board printout. The board is created out of ASCII characters, has a border around it simulating the edge of the physical board, and is printed in the centre of the terminal window. The tiles are created algorithmically, and so can form either the default board layout suggested in the game almanac, or a random pattern. Each tile consists of the number required to roll it, and a symbol representing which resource it is. If the number is 8 or 6, the number is highlighted in red to reflect the high rolling numbers. Player settlements are shown with a coloured lowercase s, player cities are shown with a coloured uppercase

`C`, and roads are shown as coloured version of the blank white lines. The robber is denoted by a lowercase `r`, and the desert tile has a desert symbol on it. Ports around the outside of the map are shown either with a `3:1` symbol representing a 3:1 wild port, or with a specific resource symbol showing that it is a 2:1 port for that resource. Due to the limitations of using an ASCII layout, the port symbols can be slightly difficult to read. However, they always signify a port at the two closest vertices.

Below the board is information about the game. In the top row, there is the current game number (if multiple games are being played), the current turn number, the current dice roll, and a display of the 6 card piles. Each pile represents one resource, or the development card deck; if the pile is empty, the symbol is not displayed. Below the top line is a numerical display of how many cards are in the resource and development card decks.

The current players are also listed, sorted by how many Victory Points they have. Each player is denoted by a coloured name and their strategy or type. In addition, MiniMax players have any Heuristic Modifiers listed after their name. The player's details also contain their resource and development card counts at the end. Any options for the player to choose or prompts for input are then displayed underneath the board, to the left of the screen.

### 4.2 Main Menu

The program contains a main menu that is displayed when the user first starts the program. This allows them to configure players and game options before the match(es) begin. These are all navigated via the keyboard, using a number selection system. The top menu (Fig. 7) is shown at first, and contains options to play the game, configure the players or the options, or exit the program. Choosing the option to configure the players will display a numbered list of the current players, along with an option to add a new player, or remove a current player by entering their number. This can be seen in Fig. 8. Finally, navigating to the 'Configure the Game Options' menu displays a numbered list of the user-configurable options from the configuration file `CONFIG.py` (Fig. 9). These can be edited by the user to change the game. However, everytime the program restarts, the default configuration is reloaded, and any user changes are lost. This menu is where the user can choose to run multiple games if they wish.

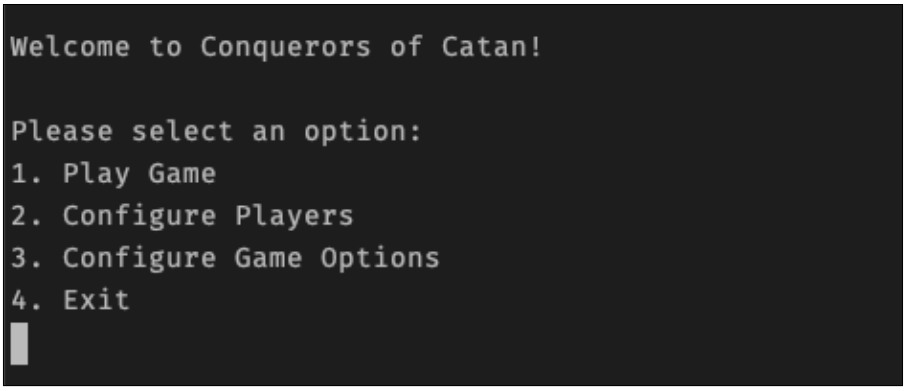


**Figure 7:** Top Level Menu

```
Please enter a player number to remove them, or choose another option:

Player 1  (AI - Random)
Player 2  (AI - Minimax [elp + wt])
Player 3  (AI - Minimax)
Player 4  (AI - Random)
5 - Add New Player
6 - Return

! - All players are AI players. The game will be for observation purposes only.
```

**Figure 8:** Players Menu

```
Select a number to modify the current setup, or return to go back:

1. Number of Matches                    10
2. Victory Point Target                 10
3. Table Top Mode                    False
4. Display Mode                       Text
5. Board Layout                     Random
6. Randomise Starting Locations      False
7 . Return
```

**Figure 9:** Options Menu

### 4.3 User Interaction during Gameplay

After the options have been configured, the main function for the user is to play the game. Each human player is assigned a player within the game by name, and their first task is to place their starting settlements and roads on the board. When a user wants to select where to place a settlement, city, or road, they input the grid reference referring to that location. An image of all the grid references can be seen in Fig. 10. It should be noted that these letters do not match the letters on tiles when playing with a physical board, as those can change between games based on the robber tiles position.

The grid reference for a location is determined by the tiles that meet at that point, such as 'a,c,e' where the tiles of 'a', 'c' and 'e' meet. If a corner of a tile does not touch any other tile, it is just given the letter of the tile and a number. As an example, the vertices of the topmost tile, starting top left and working round are 'a1', 'a2', 'a,c', 'a,c,e', 'a,b,e' and 'a,b'.


**Figure 10:** An scan of an image of the board with letters written on

The prompt a user receives to place a settlement can be found in Fig. 11. Roads are input similarly to settle-

ments, however they require two grid references, the start and end of the road. For example, a road reference could be `a1, a2` . Again, an example of the user prompt for this can be found in Fig. 12.

The author does feel that this method of entering references can be a bit cumbersome, especially for less technical users, however as the primary aim of the project was creating an AI, the author felt that this method of interaction was acceptable. This is further discussed in Section 10.2.

```
Player 1, where would you like to place your settlement?
Please enter in the form of a reference such as 'a,b,e', or of 'a1', 'a2' for single corners
(Single corner numbers increase as you move clockwise around a tile)
```

**Figure 11:** Example of prompt given to user for selecting a settlement location

```
Player 1, where would you like to place the end of your road?
Please enter in the form of a reference such as 'a,b,e', or of 'a1', 'a2' for single corners
Your road must connect to e,g,j
Please enter the 2nd location
```

**Figure 12:** Example of prompt given to user for selecting a road location, when one end has already been placed

During a human player's turn, they are presented with a series of options in the command line underneath the printed board. The first of these is simply to press enter to roll the dice, an action that must be performed at the start of every turn. After the dice is rolled, and the resource cards are dealt out, the player's resource and development cards are displayed. A set of options that they can perform are also displayed, an example of which is given in Fig. 13. These options always include viewing the building cost list, trading with a player (provided the user has at least one card to trade) and ending the players turn. Selecting one of these options will lead to further menus, all in the same style as before, either a numbered list of options or a prompt to place a structure.

```
You (Player 1) have 6 resources card(s) in your hand. They are:
1 x wheat  2 x wood  2 x sheep  0 x clay  1 x rock
You (Player 1) have 0 development card(s) in your hand.
Player 1  (Human), what would you like to do?
1: View Building Cost List
2: Trade With Player
3: Buy Development Card
4: End Turn
```

**Figure 13:** An example set of moves

### 4.4 A High Level Overview of Running the Program

Once the user has chosen to run the program, the `game` class takes over. It first performs some basic checks, such as making sure that there are the correct number of players and that they don't have the same colour or number. Then, the interface object is asked to get each player to place their initial settlements on the board. From a technical view, this is performed by the interface object calling a function in each player class to tell it to place a settlement, and the players responding by calling a function in the interface to tell it where to place it. Each player makes the intermediary decision of where to place the settlement differently (random for Random AI, or more calculated for the MiniMax AI), but they respond in the same way

After the initial placement of settlements is complete, and the resources for the second settlement have been handed out, the main game loop begins. The basic pseudocode for this can be found in Listing 1.

**Listing 1:** A pseudocode overview of the main game loop found in the play() method in src/game.py

```
while a player has not won:
    for every player:
        roll the dice
        give the appropriate cards
        while the player does not want to end their turn:
            get the interface to get them to perform a move
        if a player has won:
            end the game
```

When the dice is rolled, the interface deals out the relevant cards to the players, and the board display of the roll is updated. The roll is determined by a function that combines two random numbers between 1 and 6. This gives a probability distribution the same as the original game. The player's hand is then printed to the terminal, and the interface gives them a list of moves they can perform, and asks them to decide on one.

The method for deciding which move to perform is done differently for each player. The random player does this randomly, whereas the MiniMax player is more structured. Details of both approaches can be found in Sections 5, 6. Through their own way, the player decides on the move and then calls a function in the interface to perform it. The player can also just decide to end their turn, by raising an `end turn` error which is caught in the game loop. At the end of every player's action, a check is performed to see whether they have won, and if so the `end turn` flag is raised early, and the game is stopped.

If a player has not won by the end of their turn, some basic stats are logged such as how long they took, and how many moves they performed. Then either a short pause is called if there are no humans playing, or a longer pause is called if humans are playing to allow them to see what has happened. Finally, the loop continues and the next player's turn happens. If the match ever reaches a point where every player has had 200 turns, the match is ended early as it is likely that a stalemate has happened. In this instance, winner is calculated as the person with the most points.

When all the desired matches have finished, the `main.py` file will collect data from each of the played games to present to the user using the `tabulate` package, a fictitious example of which can be seen in Fig. 14. The calculation for average number of turns to win extrapolates for players who did not win. For example if Player

1 reached the winning 10 victory points in 30 turns, but Player 2 only reach 5 in that time, Player 2's turns to win for that game is calculated at 60.

Final Results:

| Player | Win Rate | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax) | 75.0% | 9.1 | 41 | 4.48s |
| Player 4 (random) | 25.0% | 7 | 53 | 0.03s |
| Player 1 (random) | 0% | 5 | 87 | 0.03s |
| Player 3 (random) | 0% | 4.5 | 85 | 0.03s |

Average Time per Match: 4.43 minutes

**Figure 14:** Example Stats Table after 4 matches

Finally, if more than 1 match was played, the `matplotlib` package is used to generate graphs about the matches. These include a basic score graphs, plotting each match and how many points each player got, as well as a cumulative score graph over all matches played. An example of this can be seen in Fig. 15.

### 4.5 Playing as a companion to a real board

The `table_top_mode` option within the configuration file enables the AIs to be used to play on a physical board with human players. The program is set up next to a physical board, AI players are dealt a hand on the physical board, and all human players are added to the program. Moves are performed on both boards; the AI moves are performed on the program, and then are replicated by a human on the real board, and the human moves are inputted on the program after being made on the real board.

The digital board acts as the master copy for the board. Currently, there is no way to input the tile layout into the program, and therefore the program needs to act as the master copy, deciding on the board layout. The program also needs to be the master copy as human players have the ability to interact with both boards, but the AI players do not. Inputting the tiles into the program is a future development the author would like to add, detailed more in Section 10.2.

### 4.6 Variations from the Original Game

The author wishes to mention below some differences between the digital recreation of Settlers and the original physical version. Ideally, there would be none, but due to time constraints it has not been possible to recreate the game to 100% accuracy.

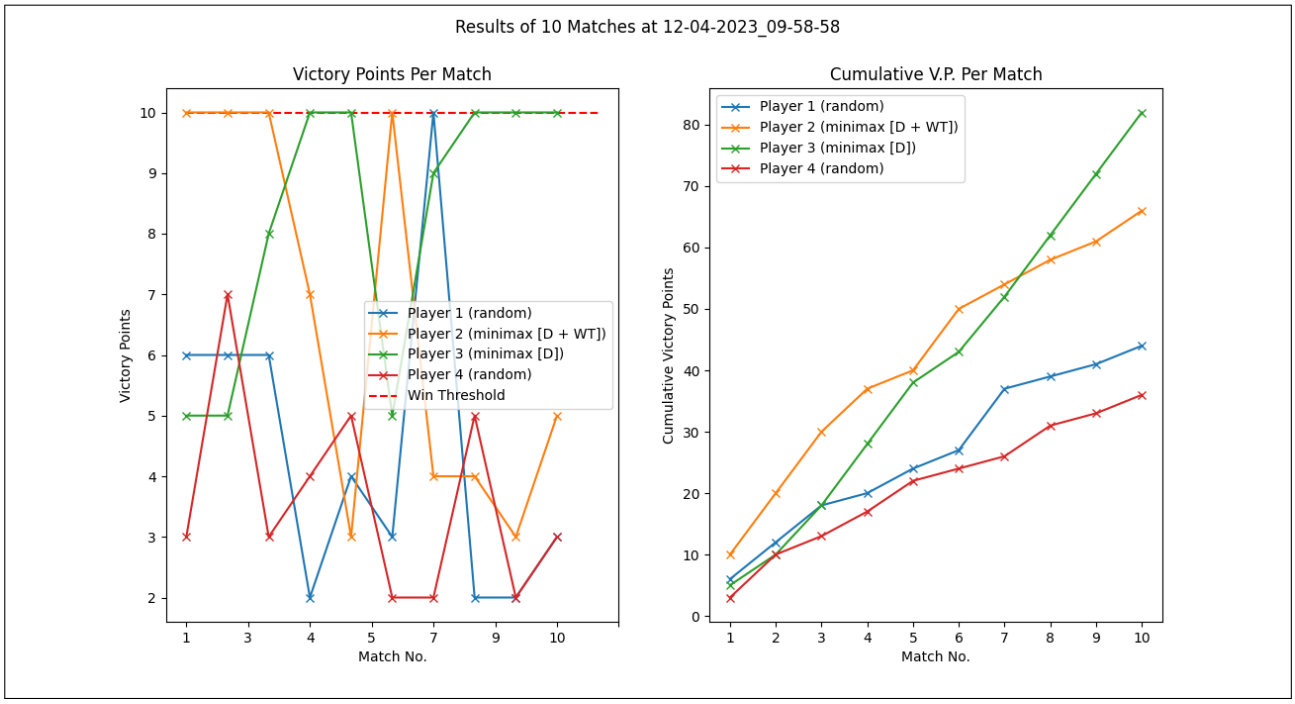


**Figure 15:** Example Graphs following 10 matches. *n.b.: MatPlotLib has caused an error in the x axis spacing in some of these graphs. The error is only in the x axis scale, the data is correct.*

- Playing cards at any time in turn:
  - In the original rules, development cards can be played at anytime during a players turn, including before the dice roll. In the digital version, development cards can only be played once the dice has been rolled. The reason for this limitation is that performing the MiniMax search before and after the dice roll would take too long computationally, and therefore it is better to limit the player in this way.
- Longest Road Interruptions:
  - As per the game almanac, the longest road can be split if another player places a settlement in the middle of it. It is then treated as two separate roads, and the longest road card is redistributed if needed. The current implementation of the longest road does not check for this, and so roads cannot be interrupted by another player placing a settlement on it.

All players are restricted by these when playing on the program.

# 5 Technical Implementation of Random AI Player

The aim of the Random AI player is simply to provide a baseline player to play against. It is not by any means designed to be a competitive or strong player, simply an opponent. In its simplest form, the player performs two main functions - choosing a move to perform, and then actually performing the move. Both of these are performed pseudo-randomly, the details of which can be found in the following sections.

## 5.1 Choosing a Move

The random player does not choose moves truly at random, more pseudo-random. By that, the author means that some moves have been adjusted in likelihood to prevent them being performed repeatedly. Settlers is a game that requires saving up resource cards over multiple turns to perform larger, more expensive moves, such as building a settlement. If the random player were to build a road everytime it could, it would never save up enough cards to build a settlement. The following table shows the adjusted probabilities for each move, in order to prevent this behaviour.

**Table 3:** Random Player Moves and Probabilities

| *Move* | *Probability* |
|---|---|
| `build city` | Default |
| `build settlement` | Default |
| `build road` | 0.5 chance if the player has less than 10 cards, else default |
| `trade with bank` | Default |
| `trade with port` | Default |
| `trade with player` | Default |
| `play development card` | 0.5 chance |
| `buy development card` | 0.5 chance if the player has less than 10 cards, else default |
| `end turn` | Default + additonal 0.25 chance after every move |

From a technical standpoint, these adjusted probabilities are coded as an additional requirement for the move to execute. For example if the move is `play development card`, an additional random 1 in 2 chance needs to be True, otherwise no move is performed. The 'Default' probability just signifies that no additional requirement needs to be met to perform the move. The author considers these modifications to the probabilities to be necessary for the random player to have some very basic strategy.

#### 5.1.1 Selection Process

The move for a random player is selected in the following way.

1. The random player requests a list of all possible moves that the player can perform from the interface object
2. The player chooses a move at random from the returned list
3. If the move requires more options, such as a settlement location, the player gets all available choices for that specific option and selects one of these at random
4. The chosen move is performed

As a player may make multiple moves in one turn, the turn is only ended either when they select the `end turn` move, or if a 1 in 4 chance comes back True. It should also be noted that the random player selects moves only from the moves it *can* perform. This means it has more strategy over a random player who is able to select moves that it can't perform.

### 5.2 Executing a Chosen Move

Most moves in the game require additional options to perform, such as specifying where a settlement should be placed, or which cards to gain from a 'Monopoly' development card. Listing 2 shows how the random player does this in a truly random way for offering a trade to a player. The `trade_with_player()` function takes 4 arguments, 3 of which need to be chosen by the player offering the trade. In order to do this, the random player gets three lists; the other players, the resources it currently has, and the resource deck. It then uses `random.choice()` to select an option from each list, and passes these into the function.

**Listing 2:** An example of how the random player selects options for a move, found in src/ai_random.py

```
if "trade with player" in chosen_move:
    # Choose a random player to trade with
    player = random.choice(
        [player for player in interface.get_players_list()
         if player != self]
    )
    # Submit the move to the interface
    interface.trade_with_player(
        self,
        player,
        random.choice(self.resources),
        # Choose a random card from the resource deck
        random.choice(interface.get_resource_deck()),
    )
```

In most cases, the options combination will always be possible, i.e. when placing a settlement, the player will select the location option randomly from all places that it legally could position the settlement. However, in a few cases, move and options combinations can be submitted that are not possible, such as trading with a

player for a card they don't have. This can result in a seemingly wasted move by the random player, however this behaviour is how a human player could perform if they don't know which cards someone else has.

### 5.3 Performance and Observations

Individual performance of the Random AI is quite basic. It's 'strategy' is mostly dependent on where it randomly places its first settlements, for example, if it places on high rolling wheat, rock and sheep locations, then on most turns it is able to buy a development card, and so it buys lots of development cards throughout the game. The author has found that it can perform surprisingly well if its random-ness leads it to build on rock and wheat, which enables it to build cities more easily, and it always builds a city when it can. However, it rarely ever performs better than the smarter MiniMax AI player, and if it does, it is more of a comment on the low performance of the MiniMax AI player, rather than the good performance of the random AI. Fig. 16 shows 25 matches played between the Random AI and an early version of the MiniMax AI. The random AI only truly beats the MiniMax player once (Fig. 16 match 9) and once by stalemate at 200 turns (Fig. 16 match 22). This statistic can be much improved when a better strategy and heuristic are applied, which the author discusses more in Section 7.

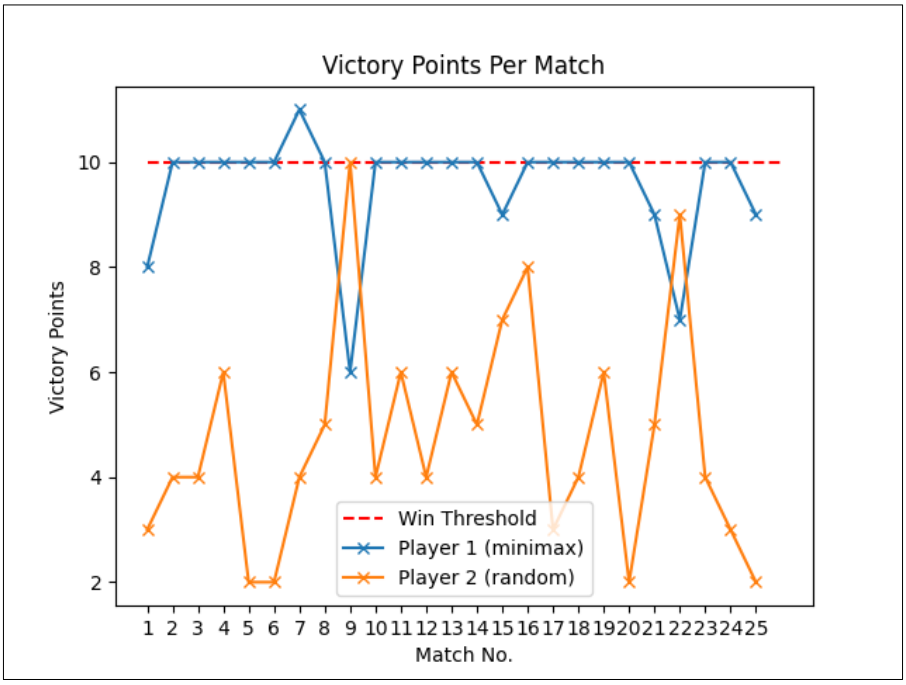


**Figure 16:** 25 matches played between Random AI and an early version of the MiniMax AI

*Comments on Fig. 16:* There are a few notable matches shown in Fig. 16. In matches 1, 15, 22 and 25, the match timed out at 200 turns, and therefore the player that currently had the most Victory Points won. In 3 cases that was the MiniMax player, but in once case (match 22) that was the random player. In match 7, the MiniMax player has 9 VPs, and then achieved either the longest road or largest army, gaining them an additional 2 VPs. This pushed them to 11 points by the end of their turn, when the game checked the VP counts and

ended. There was only one true victory for the random player, which was in match 9. After observation of other games, the author speculates that this could be as a result of the MiniMax player being fenced in by the random player, and therefore not being able to expand enough to get the winning amount of VPs.

# 6 Technical Implementation of MiniMax AI Player

The MiniMax AI player is designed to be the smart player that looks ahead multiple turns and makes moves based on where they could get to. It is not restricted to making moves just based on the current turn's information, or at random. The implementation of the MiniMax player contains two main parts, the MiniMax Search and the Heuristic Function, the latter of which is expanded on in Section 7. Section 6.3 of this chapter focuses on methods that the author used to increase the speed of the MiniMax algorithm. The speed of the MiniMax player was specifically mentioned in the project objectives.

## 6.1 MiniMax Search

### 6.1.1 In Choosing the Players Own Moves

MiniMax, as mentioned in Section 2.2, is an algorithm that searches through a tree of moves and resulting states in order to find the optimal move to take that will result in a later victory. This move may not be an obvious choice at first, but may present itself to be better as the game progresses. For example, saving cards to buy a city rather than spending them all immediately. The author had previously used a MiniMax algorithm in an earlier degree module, but [24] was used to refresh the author's memory, and provide a basic pseudocode implementation of MiniMax and AB pruning. Below is an example process of how the MiniMax search is performed.

1. Ask the interface object for a list of moves that the MiniMax AI player can make
2. Generate all possible combinations for these moves. The interface object only returns generic moves such as `place settlement`, and this step adds all possible option combinations such as `[place settlement,a1]`, `[place settlement,a2]` etc.
3. Add each of these moves to a `root score list` outside of the algorithm. This allows the score from the MiniMax algorithm to be directly assigned to the specific action, which is later used for picking the best move
4. Begin the MiniMax Search using a recursive algorithm that does the following:
    i. For each move, create identical clones of the current player and interface objects (which contains the board and other players) using `copy.deepcopy()`. Clones are made to avoid performing the move on the actual live objects
    ii. Perform the chosen move on these cloned objects using a special move interpreter that is tailored for performing moves in a simulated environment
    iii. Retrieve a new list of possible moves from this new state, increment the depth and make the next player the current player, and perform the same process over again. If the depth is at the limit, return the value of the current state (as determined by the heuristic function in Section 6.2) to the function that called it rather than performing any other moves
    iv. If all the moves under a position have been completed, return either the minimum or the maximum score to the function call above based on the level

v. If the score has been returned all the way up to the top of the branch, assign that score to the move in the `root score list` and move on to the next branch (move)

5. After the search has been performed, look through the `root score list` and find the move and option combination that results in the best score, and perform it. The move and option combination are stored as a list, with the first entry being the type of move and the rest being options. This is passed to an interpreter that then performs the move using the board interface object. If there are more than one move with the highest score, the order of preference is to build a city, to build a settlement, to build a road, and then anything else. This move order is the author's choice based on the assumption that the player will want to gain Victory Points.

Listing 3 contains a simplified pseudocode version of the code for step 4.

The author wishes to note that the MiniMax AI algorithm does assume it will only make one move per turn. This is due to computation limits, as waiting for the AI to decide it wants to end its turn during the search tree would take immeasurably more time due to the fact that most moves are better than ending the turn. This same assumption is not made when actually making moves, as the MiniMax AI, like all players, can make multiple moves per turn.

**Listing 3:** Pseudocode for the MiniMax() function in src/ai_minimax.py

```
if at top level:
    reset root score map
if minimax players turn:
    get all possible moves
    for each move:
        clone the interface and player
        perform the move using the cloned player
        if at the max depth:
            return the evaluation of the current state
        else:
            call minimax on the next players turn
        assign score value to root score map

else if not minimax players turn:
    get all possible moves
    for each move:
        clone the interface and player
        perform the move using the cloned player
        if at the max depth:
            return the evaluation of the current state
        else:
            call minimax on the next players turn
        assign score value to root score map
```

#### 6.1.1.1 Imperfect Information

The author also wishes to make a note on imperfect information. As previously mentioned in Section 2.2.4, MiniMax is usually used in a game with perfect information, and no information is hidden. This is because the MiniMax player needs to know which moves the opponent might make in order

to add them to the tree. However, in Settlers, each player's resource cards and development cards are hidden. The author has managed to work around each of these individual issues with simple reasoning.

Firstly, and most easily, the MiniMax player simply disregards development cards when considering which moves an opponent could make. These cards are fully hidden, and so the player cannot make assumptions on what they are. If the MiniMax player did assume the worst case scenario, that every development card an opponent had was a Victory Point, this would cause the MiniMax player to always be paranoid that it was about to lose, even if it was not. This would greatly affect its decision-making ability. Also, all other types of cards have options that need to be decided, such as where to place the robber, or which cards to call the year of plenty on, which would take a long time computationally. Therefore, the author decided it was better to just disregard these cards.

Secondly, the author has decided that it is acceptable for the MiniMax player to 'see' into another player's resource card hand. This is for two reasons. Firstly, if the MiniMax player wasn't able to do this, it would have to base the opponent's moves based on what they *could* potentially do from only knowing the number of cards they have. This again has similar issues to the first solution, where the number of options for this is too high to be computationally feasible. Secondly, the author believes that it is not actually difficult to know what is in a player's hand during Settlers. This is because it is public knowledge which dice number is rolled, and therefore which resource cards people are given. The author believes that someone fully dedicated to playing the game would be able to know which cards other player had by keeping track of the dice rolls. Therefore, they believe that it is acceptable for the computer to effectively keep track of cards here by just looking into a players hand when calculating what moves they can perform.

#### 6.1.2 In Response to Other Moves

The MiniMax decision process works slightly differently when responding to other moves, such as having to discard cards due to the robber being rolled, responding to a trade offer, or moving the robber. In this situation, a full tree is not constructed, rather a two layer tree consisting of the current state and all possible moves and their resulting states is made. Then, each state is evaluated, and the score that would result in is associated with the move. The move resulting in the best score is then picked. This is done for the sake of speed, as constructing a large move tree for these minor decisions is often not necessary and wastes time. Listing 4 contains an example of how this code works for discarding cards when the robber is rolled. For this example, it is assumed that the player has over 7 resources.

**Listing 4**: An example of how the MiniMax AI responds to having to discard cards, found in src/ai_minimax.py

```
# Calculate the required length
required_length = len(self.resources) // 2
while len(self.resources) > required_length:
    could_discard = list(set(self.resources))
    scores = {}
    # Evaluate the board at each possible discard
```

```
7       for card in could_discard:
8           clone = copy.deepcopy(self)
9           clone.resources.remove(card)
10          scores[card] = clone.evaluate_board(interface)
11
12      # Discard the card that results in the best board, and loop again
13      interface.return_player_card(self, self.resources[self.resources.index(min(scores, key=
            scores.get))])
```

### 6.2 Heuristic Function

The key part of any good MiniMax algorithm is the heuristic function, which is the function used to score any given game state. As mentioned in Section 2.2.2, a very basic heuristic for Settlers could just be the Victory Point score, however, in practice, a function as simple as this rarely works well. This is because as VPs increases do not happen every turn, and so a player could be choosing blindly between moves because they all don't increase the VP score.

Through iterative testing, the author has come up with the heuristic listed below in Table 4 to act as a good starting point for a MiniMax player. The basic format for the scoring a position using this function is twofold. Firstly data is collected about the player, including information such as their buildings positions, whether they are in the lead and which resources are in there hand. Secondly, this data is passed to the basic heuristic function which converts it into a score as detailed in the table. If the player has the required number of VPs to win, it returns 1,000,000, the highest score it can. If the player has lost, as another player has the required number of VPs to win, it returns -1,000,000. The 'Roll Map' mentioned is further explained in Section 6.2.1.

**Table 4:** Main MiniMax Heuristic as of 18 April 2023

| *Stat Collected* | *Score Modification* | *Further Explanation (If Required)* |
|---|---|---|
| Player Victory Points | + (10 * Number of Victory Points) | |
| — " — | + 10,000 if one VP away from the target score | |
| — " — | + 100 if the player has the most VPs | |
| — " — | + 50 if the player is tied for the most VPs | |
| Player Settlements | + 500 Per Settlement | |
| — " — | + 2 * Sum of Tile Frequency that the building is on | Rewards going for tiles with higher frequency rolls |
| — " — | + 2 * No. Tiles the building is on | Rewards placing buildings on junctions with more tiles |
| Player Cities | + 1000 Per City | |

| *Stat Collected* | *Score Modification* | *Further Explanation (If Required)* |
|---|---|---|
| — ” — | + 3 * Sum of Tile Frequency that the building is on | Rewards going for tiles with higher frequency rolls |
| — ” — | + 2 * No. Tiles the building is on | Rewards placing buildings on junctions with more tiles |
| Player Settlements and Cities | + 10 if the average distance between buildings < 6 | Rewards placing settlements somewhat together. Only takes affect during placement phase. |
| — ” — | - 10 if the average distance between buildings > 8 | Penalises placing settlements far apart. Only takes affect during placement phase. |
| Player Ports and Roll Map | If 3:1 Port, + 3 * highest value in Roll Map | Rewards having a 3:1 port in conjunction with a high rolling resource |
| Player Ports and Roll Map | If 2:1 Port, + 2 * resource score in Roll Map | Rewards having a 3:1 port in conjunction with a high rolling resource |
| Player Resources in Hand | + Amount of Resources | |
| — ” — | - 0.5 * Count of resources over 12 | Penalises hoarding resources. 12 was chosen based on the author’s observation, although 7 could be justified as well based on the robber. |
| Resources Player can Access | + 3 * Amount of Resources player has access to from dice roll | Rewards trying to access all 5 resources |
| Special Cards | + 50 if player has the largest army | |
| — ” — | + 50 if player has the longest road | |
| — ” — | - 25 if player has played > 2 robbers more than the next player | Penalises playing more robber cards if the player has the largest army |
| Length of Longest Road | + 2 * Longest Road size if the player does not have the longest road, else 2 | Rewards going for the longest road, but penalises adding to it if the player already has it |
| Player Development Cards | - 5 * No. VP Development Cards | Penalises relying on VP Cards |

| *Stat Collected* | *Score Modification* | *Further Explanation (If Required)* |
|---|---|---|
| — ” — | - 0.5 * Count of Development Cards over 12 | Penalises hoarding Development Cards |
| Player Roads | - 1.5 * Count of Roads over 10 | Penalises road spamming |
| — ” —- | + 25 if roads / No. Buildings <= 2 | Basic calculation to evaluate whether the buildings are placed with the minimum spacing between them to create more placement locations |
| Available Build Positions | + 5 * Available Build Positions | Promotes building in ways that create settlement positions |
| Opponents on Road | - 5 * Number of Opponents on Road | Penalises building where an opponent could also build |

The heuristic function above returns the score, as well as a `score_map`, which is a key-value dictionary showing how it has calculated the score. This is used by the more advanced strategies discussed later in Section 7.

Table 6 in Section 7.2 shows some tests matches played between the MiniMax player with no strategy, and 3 random players, for the purpose of exploring the Wishful Thinking modification. As can be seen, using only this base heuristic wins just 2/3 of games, and takes around 109 turns to do so on average. The author believes that this can be improved upon. Some general observations on behaviour can also be found in Section 9.3.

#### 6.2.1 Roll Map

One key stat used both in the base heuristic, and in the more advanced developments (Section 7), is the ‘Roll Map’. The Roll Map indicates how likely a player is to gain a particular resource on any dice roll, where the higher the score, the more likely they are to get it. The score for a resource is based on how many cities or settlements a player has on those resource tiles and their frequency (a list of tile frequencies based on dice roll are shown in Table 7, Appendix A).

In order to calculate, the following is done:

- For each settlement, the frequency of every nearby resource tile is added to the roll map.
- For cities, this frequency is doubled.

A worked example can be seen in Fig. 17.

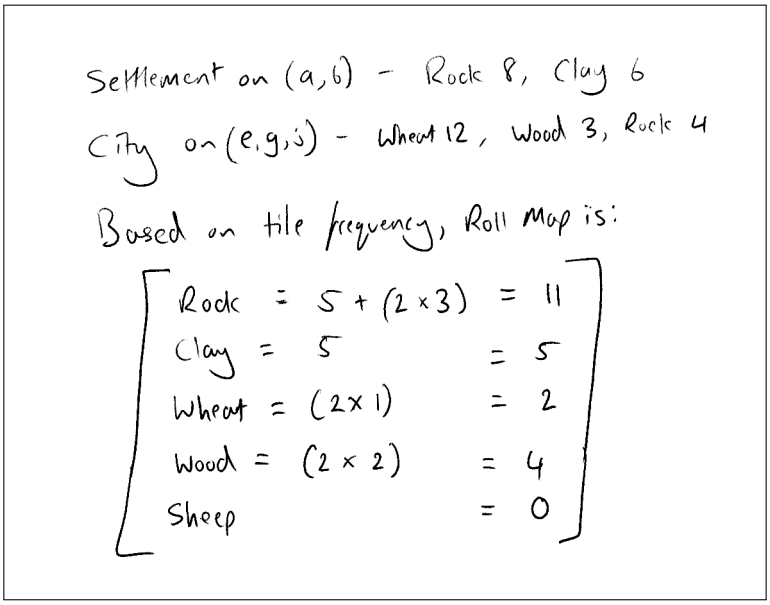


**Figure 17:** An example Roll Map calculation, showing buildings and their tile numbers, and the resulting score in the Roll Map after converting the dice roll to a frequency

## 6.3 Increasing the MiniMax Players Speed

Originally identified in the project outline is the aim that the MiniMax player's turn speed should be relatively fast, otherwise the human player may lose interest. The author believes that an AI player that performs 90% 'good' moves and takes 10 seconds is (overall) better than one that performs near 100% 'good' moves, but takes 60 seconds. Therefore, below are listed some ways that have been tested and/or implemented in order to increase the MiniMax AI players speed.

### 6.3.1 Alpha-Beta Pruning

As mentioned in Section 2.2, AB Pruning is a well recognised method used to speed up the MiniMax search algorithm by decreasing the number of nodes that need to be evaluated. In practice, this is quite a simple implementation, and the code shown in Listing 5 paired with passing the current `alpha` and `beta` values to each MiniMax function call is all that is needed to implement this. No speed test data was recorded during the addition process, however a clear reduction in computation time was observed, and the author estimates this to be around 1/2.

**Listing 5:** Alpha-Beta Pruning code found in src/ai_minimax.py

```
1
2  # On the AI player's turn:
3
4  alpha = max(alpha, max_combo[1])
5  if beta <= alpha:
```

```
6       self.log(f"Pruning at depth {max_depth}")
7       break
8
9   # On the opponents turn:
10
11  beta = min(beta, min_combo[1])
12  if beta <= alpha:
13      self.log(f"Pruning at depth {max_depth}")
14      break
```

#### 6.3.2 Epsilon Pruning

Epsilon Pruning (EP) is a method created by the author to prune a MiniMax tree during its formation stage, in order to limit the total number of nodes that need to be evaluated. In a game like Settlers, there are often many similar moves that can be made by a player, e.g. Trading Rock for Wood, Rock for wheat, Rock for Clay, or Building a Road at any number of road endings that the player has. This leads to an exponential increase in the number of node evaluations required for MiniMax search, as for every node added, its entire subtree needs to be explored.

EP is devised to combat this in the following way: When appending all move and option combination to a list to be looked through (as mentioned in Section 6.1.1), all the moves of the same type are evaluated (e.g. placing a road), and only add the move that returns the highest score. This greatly decreases the number of evaluations required during the search, and thus the time taken. However, there may be a small decrease in accuracy, as a potentially promising move might be skipped if another node appears to be better, but actually proves not to be fruitful.

To help mitigate this performance drop, EP has two different levels. Level 1 only evaluates and reduces what the author considers to be *unimportant* moves, such as offering trades, or choosing combinations for development cards. It does not include the more important moves such as road and settlement placements, which are instead included in Level 2. This higher level (2) provides a large increase in speed compared to not using EP, but does also entail a slight decrease in accuracy, as much fewer move paths are considered.

The author notes that after applying all the other speed increase methods, EP is usually not necessary. However, EP will be useful when exploring the MiniMax player to much deeper game tree levels.

#### 6.3.3 Copy Operations

In order to perform all possible moves without modifying the current game state, the MiniMax player performs multiple millions of copy operations per game. The author originally used the `copy.deepcopy` method to perform these, which was chosen as it fully copied both the immediate attributes and the connected objects, whereas `copy.copy` performs a 'shallow' copy, just copying the immediate class attributes and using references for the connected pre-existing objects. However, this was found to be taking around 80% of the

overall program run time to perform (as can be seen in the top row of Fig. 18) and therefore was an obvious place to try to improve the speed of the algorithm.

The author decided on using a serialisation [25] method to clone the objects, and ran multiple tests to evaluate the fastest one. The methods included in the tests were:

- `copy.copy()`
- `copy.deepcopy()`
- `json.dumps()` and `json.loads()`
- `ujson.dumps()` and `ujson.loads()` [26]
- `pickle.dumps()` and `pickle.loads()`,
- `jsonpickle.encode()` and `jsonpickle.decode()` [27]
- `marshal.dumps()` and `marshal.loads()`

These options were discovered from a Stack Overflow thread from a user who had a similar issue [28]. `copy`, `json`, `pickle` and `marshal` are all builtin Python modules, whereas `ujson` and `jsonpickle` are third party. The full testing method can be found within `src/dev/timings.py`.

Some methods did not know how to encode the `logger` objects used to conduct logging in the players, and so could not be used, but of the remaining methods that did work, `pickle` was the fastest. Therefore, the default implementation of `deepcopy` was overwritten in all players classes, as well as the interface class, to use the `pickle.dumps()` and `pickle.loads()` combination, and the `__eq__()` method used to determine whether two objects are equal was also overwritten. This lead to a large decrease in the amount of time taken to perform these copy operations (which can be seen in rows 1 and 3 of Fig. 19), and therefore an overall speed increase of around 50% for the MiniMax AI player.

**Listing 6**: Overwritten __deepcopy__() and __eq__() methods in src/ai_player.py

```
# Overwritten methods for faster copy operations

def __deepcopy__(self, memodict={}):
    return pickle.loads(pickle.dumps(self, -1))

def __eq__(self, other) -> bool:
    if not isinstance(other, player):
        return False
    return self.number == other.number
```

#### 6.3.4 Parallelization

The MiniMax AI players speed is further increased from a technical instead of an algorithmic point due to the addition of parallelization. Parallelization is the process of running multiple operations at once, and in this instance is implemented using the `concurrent.futures` multiprocessing module [29], which changes the MiniMax AI player to perform the search across all the cores of the CPU.

Statistics Call Graph

| Name | Call Count | Time (ms) | Own Time (ms) |
|---|---|---|---|
| deepcopy | 247839903 | 594984 82.1% | 287212 39.6% |
| <method 'get' of 'dict' objects> | 516978826 | 54395 7.5% | 54395 7.5% |
| <built-in method builtins.id> | 329025199 | 35006 4.8% | 35006 4.8% |
| <built-in method time.sleep> | 67 | 31698 4.4% | 31698 4.4% |
| _deepcopy_atomic | 186818145 | 22591 3.1% | 22591 3.1% |
| _keep_alive | 35656750 | 28922 4.0% | 20697 2.9% |
| calculateVictoryPoints | 578427 | 34005 4.7% | 17709 2.4% |
| _reconstruct | 4299926 | 593050 81.9% | 12995 1.8% |
| update_special_cards | 82058 | 23959 3.3% | 11297 1.6% |
| __eq__ | 35098815 | 14294 2.0% | 10648 1.5% |

**Figure 18:** Top 10 Rows of Code Profiling as of 11 April 2023

Statistics Call Graph

| Name | Call Count | Time (ms) | Own Time (ms) |
|---|---|---|---|
| <built-in method _pickle.loads> | 1710800 | 191725 15.0% | 168636 13.2% |
| calculateVictoryPoints | 5059593 | 318515 24.9% | 165891 13.0% |
| <built-in method _pickle.dumps> | 1710800 | 191118 14.9% | 163812 12.8% |
| evaluate_board | 791518 | 594477 46.5% | 113961 8.9% |
| __eq__ | 334916539 | 144208 11.3% | 108162 8.5% |
| get_roads_list | 353509821 | 57268 4.5% | 57268 4.5% |
| update_special_cards | 348565 | 137573 10.8% | 56220 4.4% |
| <built-in method builtins.isinstance> | 368705942 | 40605 3.2% | 40604 3.2% |
| return_clusters | 2185778 | 51940 4.1% | 38314 3.0% |
| find_route_from_node | 10665423 | 53194 4.2% | 37203 2.9% |

**Figure 19:** Top 10 Rows of Code Profiling as of 13 April 2023

In practice, this takes place when the MiniMax function is first called. The function detects that it is the first call and so is at the top of the tree, and instead of executing the move and calling itself like normal, it sets up a `ProcessPoolExeuctor`. It then distributes each potential move to this pool, which in turn assigns it to a core to be evaluated. This is essentially the same as increasing a one lane road to an $n$ lane road, where $n$ is the number of cores the CPU has. The effective speed increase here is variable based on the number of moves being executed, and how long they each take, but assuming they all take the same time, the following formula can be used to represent the execution time of $x$ tasks:

$$x/n + (x \bmod n)/n \tag{1}$$

where $x$ is the number of tasks, and $n$ is the number of cores.

Therefore, 10 tasks over 1 core would take time of 10 (10 tasks done sequentially), whereas 10 tasks over 4 cores would take time of 3 (4 tasks in parallel, then another 4 tasks in parallel, then 2 tasks in parallel).

It should be noted that parallelization is not always used. In the instance where there is only 1 move, it takes longer to create the pool, submit the task and fetch the result than it does to just run the single task. It should also be noted that the overall speed increase is affected by the computer the program is running on - a computer with 4 cores will benefit from around half the speed increase that a computer with 8 cores benefits from this.

#### 6.3.5 Speed Increase Summary

Overall, the resulting speed increase from both the high level algorithmic improvements, and low level technical improvements is very significant. Originally, the MiniMax player could take multiple minutes on some turns to reach a decision, sometimes even more than 10 minutes. After the implementation of all of these features, the average turn time for the MiniMax player is just 4 seconds, putting it well under the baseline 15 seconds outlined in the project objectives, and comfortably under the ideal time of 5 seconds.

### 6.4 MiniMax for Greater Than 2 Players

As per the original rules [3], Settlers is not meant to be played with less than 3 players, however, MiniMax (in its original form) is not designed to be used in games with more than 2 people (Section 2.2.4). Therefore, the author's original software implementation was modified and extended to create a new form, which they refer to as MiniMiniMax (or technically, $Mini^{n-1}Max$, though this is far less catchy).

MiniMiniMax is the same as MiniMax, but scaled up for more players. In a game of 3 players, and a tree height of 4, the top level will be the current player, the next layer will be player 2, the third player 3, and then back to the first player. At any level that is not the current player, the algorithm still assumes the opposing player will perform the move that will be the worst for the current player. However, as detailed in [30], this can lead to somewhat pessimistic planning, as the MiniMax player assumes that every opponent is teaming

up against them. In practice, this is not often too detrimental, as Settlers is a game that does not provide many opportunities for one player to directly target and hinder another.

Other extensions of MiniMax, such as MaxN [31], might provide improvements to the author's extension, by instead assuming that other players will pick the most that is best for them. The author did try to implement MaxN, and this is discussed in Section 10.2.3. For the remainder of this report, the author refers to a MiniMiniMax player when using the term MiniMax.

# 7 Strategies and Heuristics Research

## 7.1 Introduction

While the performance that the default heuristic performs is good, the author believes it can be improved through the process of giving the MiniMax player more of a specific, advanced strategy. The method for adding these strategies utilises the `score map` previously identified in Section 6.2. The map consists of key-value pairs showing why and by how much the score was modified by one of the statistic items found in Table 4. It acts similar to a recipe, or method, showing how the heuristic function achieved it's score. An example can be seen in Listing 7. Other heuristic functions work by building on top of this score map, amplifying, reducing, or adding new entries to it to modify the final score.

**Listing 7:** Extract of a score_map, from src/heuristic_modifiers.py

```
score_map = {
    "victory points" = 70,
    "leading" = 100,
    "relying on dev cards" = -10,
    ...
}
```

Due to their nature, it is also possible to apply multiple other heuristic functions on top of the original one, as 'building blocks', to create a more complex compound strategy. These can be smaller, situational tactics, such as aiming for wood and clay if it's within the first few turns of the game, or they can be larger, more sweeping modifications, such as heavily discouraging acquiring ports throughout the whole game.

Below are some of these strategies that the author theorised and tested, alongside how they performed. As there were many strategies to investigate, it was not feasible to perform human testing on them, and so the testing procedure was simply to play a set number of games using a random board layout, 3 random players, and a MiniMax player. It was also not possible to complete full testing of all combinations, and so often a strategy was either tested in isolation, or with the current best set of strategies. All strategies also utilised the Wishful Thinking Modification, described more in Section 7.2. The best strategies and final evaluation of these can be found in Section 9.

## 7.2 Wishful Thinking Modification

The author has theorised that the MiniMax player's ability to plan long term is limited due to its nature of assuming it won't gain any more resources in future turns. Early games suggested that the MiniMax player would move down the search tree onto future turns assuming that no further resources would be given, rapidly depleting its hand. The author believes that this leads to a somewhat conservative player who does not aim high; they are only focused on what they can do in the short term with their current resources.

This was proven when code was added to the MiniMax player class that injected cards into the players hand for future turns, simulating the dice roll and gaining cards. This is done at a rate of $\{a + x \mid a \in R\}$ where

$x$ is the number of players / 2, $a$ is the resource and $R$ is the player's resource set. The author believes this rate is a fair estimate, as it is based on the number of players (which itself reflects the number of dice rolls between each MiniMax player's turn), and the resources the player actually has access to. The author names this "Wishful Thinking (WT)".

**Listing 8:** Wishful Thinking Code found in src/ai_minimax.py

```
if self.wishful_thinking:
    for card in self.has_access_to(interface_clone):
        for _ in range(math.floor(len(interface_clone.get_players_list()) / 2)):
            player_clone.resources.append(card)
```

In practice, this modification benefits the MiniMax player greatly. Fig. 20 contains two Cumulative VP Graphs (with different scales) showing the results of 50 matches played between 3 random players and a player either with or without Wishful Thinking. The player with WT gained a higher average number of points per turn, and finished the 50 matches with around 30 more VPs. Table 5, 6 display more statistics, going to show that the MiniMax player *with* WT had a higher win rate, and on average took 30 turns less to win a match. The author theorises that these improvements are a result of the player being able to plan further ahead, acting more efficiently, and therefore requiring less turns to win.

As a second benefit, this implementation solves the issue of random chance identified in Section 2.2.4. As the cards are injected based on what the player has access to, and the number of dice rolls between the MiniMax players turn, this aids the random chance element of the game by using statistical probability to assume what cards may be given.

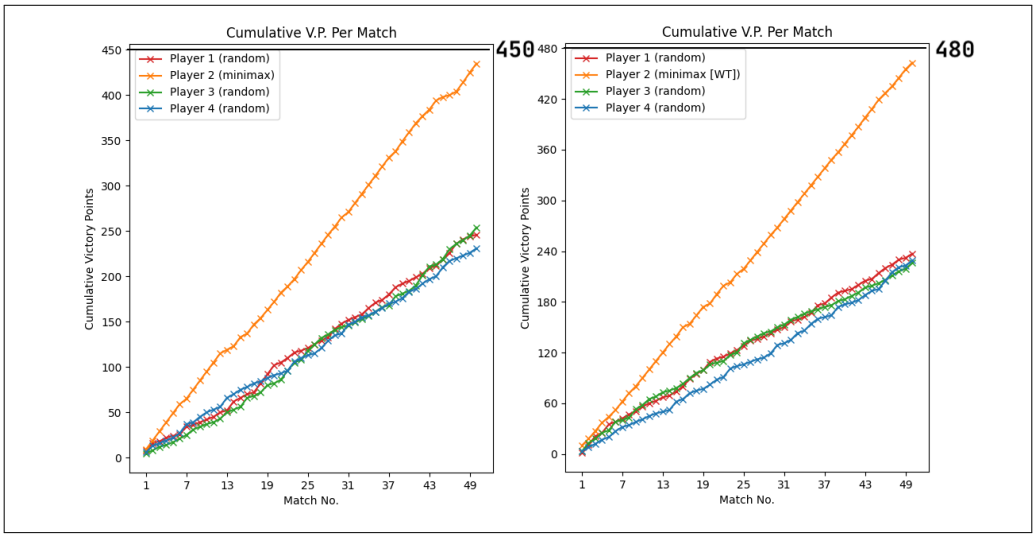


**Figure 20:** Cumulative VP Graph of 50 matches played between 3 random players and a MiniMax player either *without* (left) or with (right) Wishful Thinking. *n.b.: Graphs have different scales*

**Table 5:** Statistics of 50 matches played between 3 random players and a MiniMax player *without* Wishful Thinking

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax) | 66% | 8.7 | 109.018 | 4.43s |
| Player 3 (random) | 14% | 5.08 | 166.882 | 0.03s |
| Player 1 (random) | 12% | 4.92 | 172.45 | 0.03s |
| Player 4 (random) | 0% | 4.62 | 183.19 | 0.03s |

**Table 6:** Statistics of 50 matches played between 3 random players and a MiniMax player *with* Wishful Thinking

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [WT]) | 72% | 9.26 | 88.076 | 6.61s |
| Player 4 (random) | 12% | 4.58 | 168.592 | 0.03s |
| Player 1 (random) | 8% | 4.74 | 160.665 | 0.03s |
| Player 2 (random) | 8% | 4.52 | 162.42 | 0.03s |

Interestingly, the player with WT took longer to make moves on average, and the matches involving this player also took longer on average. This is because the player has more options to consider as a result of having more cards, and therefore it takes longer to evaluate all the possible options.

Overall, the author believes that the WT Modification is a useful and successful addition to the player that suitably increases its performance based on realistic assumptions. While it does increase the average calculation time to longer than the ideal of 5 seconds, the author believes that the performance increase and turn limit decrease are great enough to negate this. This is further confirmed by the fact that any respectable Human Player would typically take longer than the MiniMax players new average of 6.61 seconds per turn. The author's father recalls a player who would often take 15 minutes to place their first settlements.

### 7.3 Notable Heuristic Modifications and their Performance

#### 7.3.1 Clay and Wood

Clay and Wood is a well-used strategy when playing Settlers, whereby a player aims to gain as much wood and clay as possible to build roads, and in order to help build settlements. As clay is often a more scarce resource,

this does also give the player a certain amount of bargaining power in trades too. The implementation of this strategy uses the template modifier `HMFavourResources` to increase the score if the player holds these cards in their hands, or if they are built on tiles that produce them (utilising the Roll Map). This can be seen in Listing 9.

**Listing 9:** Code from the HMFavourResources class within src/heuristic_modifier.py

```
for resource in self.resources:
    mod_map[resource] = stats_map["resources"].count(resource) * 2
added = 0
for resource in self.resources:
    if resource in stats_map["roll map"]:
        added += stats_map["roll map"].get(resource)
mod_map["favoured_resources"] = added * 3
```

In testing, this strategy does not perform too well. As seen in Table 8, it only won 70% of the 10 games it played, and had an average turns-to-win of 98.936, much higher than some other strategies tested. The author thinks this could be because this strategy is only useful for building roads and settlements, which is not enough to win the game. Therefore, a strategy for the late game is required in order to carry through the head start this has in the early game.

#### 7.3.2 Rock and wheat

Rock and Wheat is another well-used strategy from the Settlers community, with the aim being to utilise these to build as many cities as possible. The strategy again utilises the template modifier `HMFavourResources`, seen in Listing 9, but this time for Rock and Wheat.

The strategy by itself is fairly mediocre. In 10 games, the player using the strategy won 90%, with an average VP score of 9.6. It took 69.508 turns on average to win, and had an average turn time of 8.56s, which is one of the slowest in all the strategies. Full results in Table 9. This might prove to be a fruitful end game plan when used in conjunction with the Clay and Wood strategy, but this was not tested as such. The player would need to decide which strategy used based on either which turn it is on, or how many settlements and cities it has.

#### 7.3.3 Development Cards

The 'Development Card Spam' strategy is one that contains two modifications to the score. The first adds a new score based on the Roll Map scores for sheep, rock and wheat, in order to promote it placing on those. This is similar to an in-built Favour Resources modifier. The second is a new statistic that increases the score the more development cards that have been played. The code for this can be seen in Listing 10.

**Listing 10:** Code from the HMDevelopmentCardSpam class within src/heuristic_modifier.py

```
added = 0
for resource in ["sheep", "rock", "wheat"]:
    if resource in stats_map["roll map"]:
        added += stats_map["roll map"].get(resource)
mod_map["sheep_rock_wheat_rollmap"] = added * 3
mod_map["total_played_development_cards"] = stats_map["total_dev_cards_played"] * 3
```

This strategy has initially proven to be one of the best performing strategies out of all that the author has tested. It won 100% of the 10 test games it played, with an average VP count of 10.2, and an average turns-to-win of 55.3, which is the lowest seen so far. The full results can be seen in Table 10. The author believes this strategy performs well due to its simple and easy nature of aiming for one goal, which is to gain development cards. The strategy also contains prioritising rock and wheat, which are used for cities, and so the player is also more easily able to build these, which are key to winning. In a later game against 2 random players and 1 human, the MiniMax player lost from a winnable position when the Longest Road and Largest Army cards were taken from it in the mid to late game. This identifies one weakness in this strategy that may not have appeared when testing against random players.

#### 7.3.4 Early Expansion

Another strategy that performed notably well is the 'Early Expansion' strategy. This strategy encourages the player to expand and build roads, and to do so in a way that increase the number of settlement positions available. However, it is only applied while it is either within the first 15 turns of the game, or while the player has less than 7 roads.

**Listing 11:** Code from the HMEarlyExpansion class within src/heuristic_modifier.py

```
if interface.turn_number < 15 or len(stats_map["roads"]) < 7:
    mod_map["roads"] = 7 * len(stats_map["roads"])
    if "available_settlement_positions" in mod_map:
        mod_map["available_settlement_positions"] = (
            2 * mod_map["available_settlement_positions"]
        )
```

In testing, this strategy performed almost identically to the Development Card strategy, winning 100% of the 10 games they played, and finishing with an average of 10.2 VPs per match. However, this strategy had a slightly higher average turns-to-win of 59.6. Full results in Table 11.

#### 7.3.5 No Ports

One controversial idea that the author tried was a strategy that heavily discouraged the claiming and usage of ports. This was done simply by reducing the score of a game position by 50 per port that the MiniMax player owned, as seen in Listing 12. This is a controversial strategy as ports are often useful, but the author wanted to see if the player would fare better without them.

**Listing 12:** Code from the HMIgnorePorts class within src/heuristic_modifier.py

```
for key in list(mod_map.keys()):
    if "port" in key:
        mod_map[key] = -50
```

In practice, this strategy did not provide any sort of performance gain. Table 12 shows that the win rate was only 70%, and that the average turns to win was 95.451, both of which are quite poor. The author believes this may not necessarily be just due to the strategy itself, but might also be as a result of its implementation. During observations of these games, the player was still building on ports, although only after it could not build on other spots. This is because the penalty of -50 is only enough to make it change to a different, non-port location, not to stop it building there altogether. The author wishes to test this further with a stronger penalty, to see if that changes the outcome.

## 7.4 Compound Strategies

### 7.4.1 Early Expansion and Favour Resources (Rock and Wheat)

When compounded the somewhat mediocre Rock and Wheat strategy is combined with the `EarlyExpansion` strategy, the performance is much better. As can be seen in Table 14, while the win rate *doesn't* improve, the average victory points, turns-to-win, and turn time all do. The turns-to-win decrease by 10 to only 59.078, and the average victory points increases to 10.1. The turn time also decreases to only 6.49s as opposed to the previous 8.56s. This improvement is potentially because the player utilises the Early Expansion strategy to claim a large area at the start of the game, giving itself many places to build cities later on. Based on this, the author believes that the original Rock and Wheat strategy fails due to it only going for rock and wheat, and so not being able to expand enough during the early game in order to actually have places to build cities in the late game.

### 7.4.2 Early Expansion and Favour Resources (Clay and Wood)

Interestingly, when the Early Expansion strategy is combined with the clay and wood variant of the `HMFavourResources` strategy, it performs worse than without it. It wins only 80% of games, and takes a much longer 91.8 average turns-to-win (Table 13. The author theorises that this is because the `HMFavourResources` strategy heavily influences settlement position away from tiles that generate rock and wheat for cities. This limits the late game performance of the player, as it is unable to easily build cities. The author believes that a more fine-tuned version of this, which is only applied while the `HMEarlyExpansion` is applied may yield better results.

# 8 Testing Procedure

Testing was conducted by the author to find out both functional errors that were within the program, and UX improvements that were noticed and improved on.

## 8.1 Functional Testing

The process for bug testing was iterative, and happened in all phases of development. At periodic intervals, the author would run the game, typically between two AIs, and observe the behaviour. Two types of errors were observed, the first was errors relating to the rules of the game, and the second was programming errors. Detailed below are examples of both.

For a large part of development there was an error where the longest road was not calculated properly. In the rules, the longest road is calculated as 'the longest single chain of continuous pieces', however in the program this was simply calculated as the largest cluster of roads that a player owned. This was a known bug, an example of a game rule error, and one that was later fixed using an algorithm generated by ChatGPT.

The other type of error is a programming error. For example, a bug appeared where in certain situations, receiving a development card would cause the program to crash. This only happened when the development card pile was empty, and so was quite rare to see as it happened far into the game. Eventually this was caught by logging which card was being given to the user, and it was noticed that when the pile was empty, `False` was being returned to the user, which was causing the error.

Another programming error was cards going missing. The interface object was designed to create a standardised method of giving and receiving cards in order to avoid players handling their own giving and receiving of cards. It was observed that as games went on, the total number of cards in the game was decreasing, so that the total number of resource cards was no longer 95. After much logging, and the creation of the `verify_game_integrity()` function within the `board_interface` class, it was discovered that the issue was happening when a card was being stolen from a player as a result of the robber. The stolen card was being removed with a `pop()` call, however this was not assigned to anything, and so the card was just removed from the player's resources, and not returned to the board. Then a different card of the same type was being given from the board to the player. Therefore, a card disappeared everytime this happened. This was fixed simply by replacing the card given from the board with the card taken from the player.

The player and interface classes all have loggers, which meant that most of the bugs were caught using log and print statements. The more major bugs that were not fixed immediately were added to Jira, an example of which can be seen in Fig. 21.

## 8.2 UX Testing

UX Testing was conducted similarly to bug testing, but instead of the author observing the game played between two AIs, the author would instead also play. This enabled the author to discover some smaller 'non-

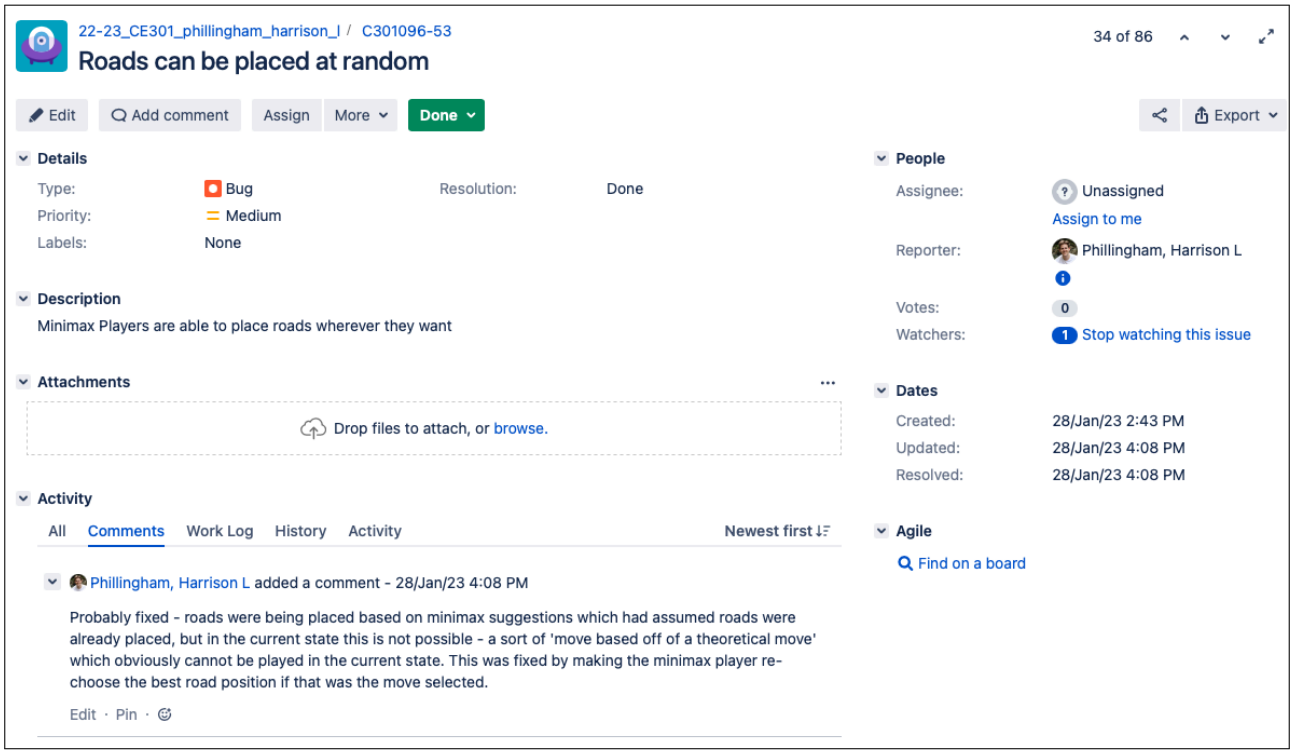


**Figure 21:** An example of a bug reported to JIRA

breaking' changes that needed to be made in order to improve the experience of human players. For example, the original method for choosing a move on a human players turn was to input the entire move, such as `build settlement` for building a settlement. However, through testing, the author realised that this was not a good way to interact, as spelling mistakes could occur, and it was sometimes ambiguous as to what the correct prompt should be. Therefore, the author move to using the number system now seen in the project, an example of which can be seen in Fig. 13.

# 9 Performance Evaluation and Comparison of AIs

## 9.1 Comparison Against Random AI

The author ran 50 games with the 3 best strategies to get a more accurate level of their performance, and to find the overall best strategy. The tests were conducted on random boards, with the AI MiniMax player and 3 random players, and the results are included below. All three strategies tested won 100% of the games, and so the only main differentiating factor is how fast they did this, or their average turns-to-win.

### 9.1.1 Evaluation of Early Expansion Strategy

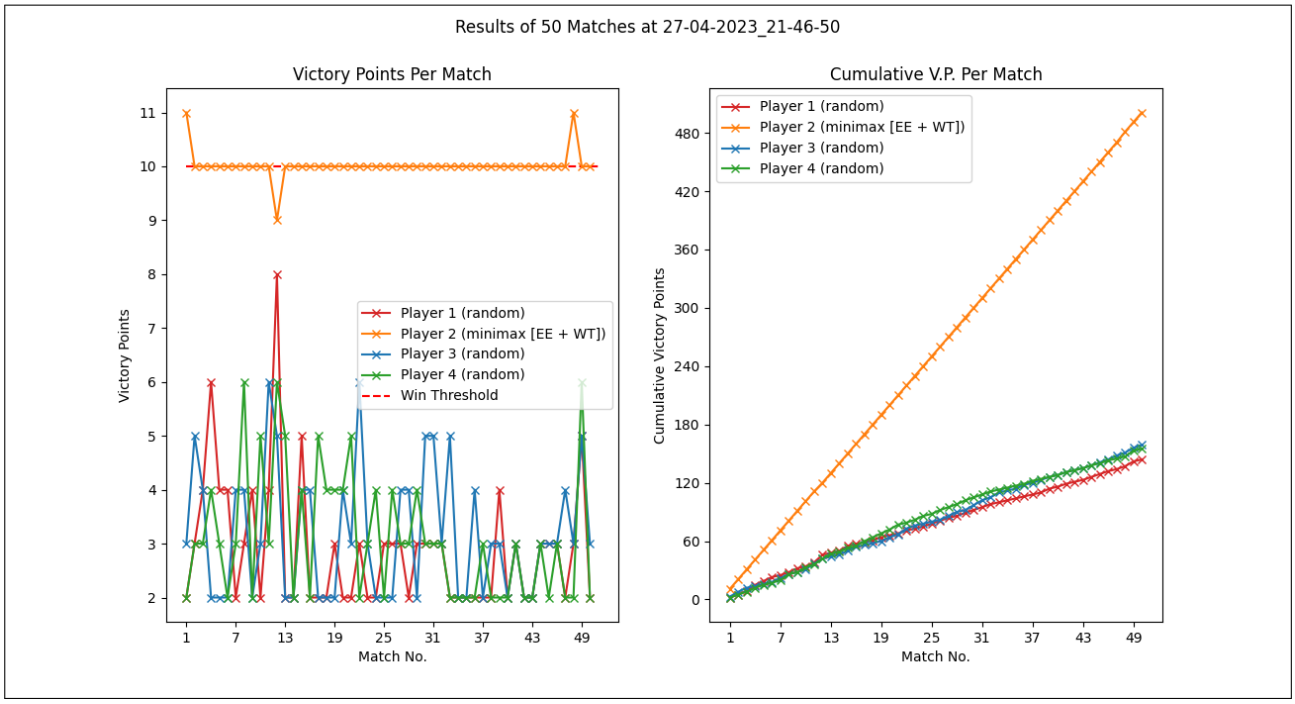


**Figure 22:** Evaluation results of the Early Expansion Strategy

The least well performing of the 3 best strategies is the Early Expansion strategy. Table 15 shows that while this did win all games played, it did so in a longer time, at 44.54 average turns-to-win. This is because the Early Expansion strategy is only applied during the first 15 turns of the game, or while the player doesn't have many roads. Therefore, as soon as neither of these conditions are met, the player has no strategy other than the default heuristic. This strategy does have a surprising impact on the player given how it only affects the player's action for around 1/3 of the turns, however the author feels this strategy is best used in conjunction with another, to provide a complete game plan.

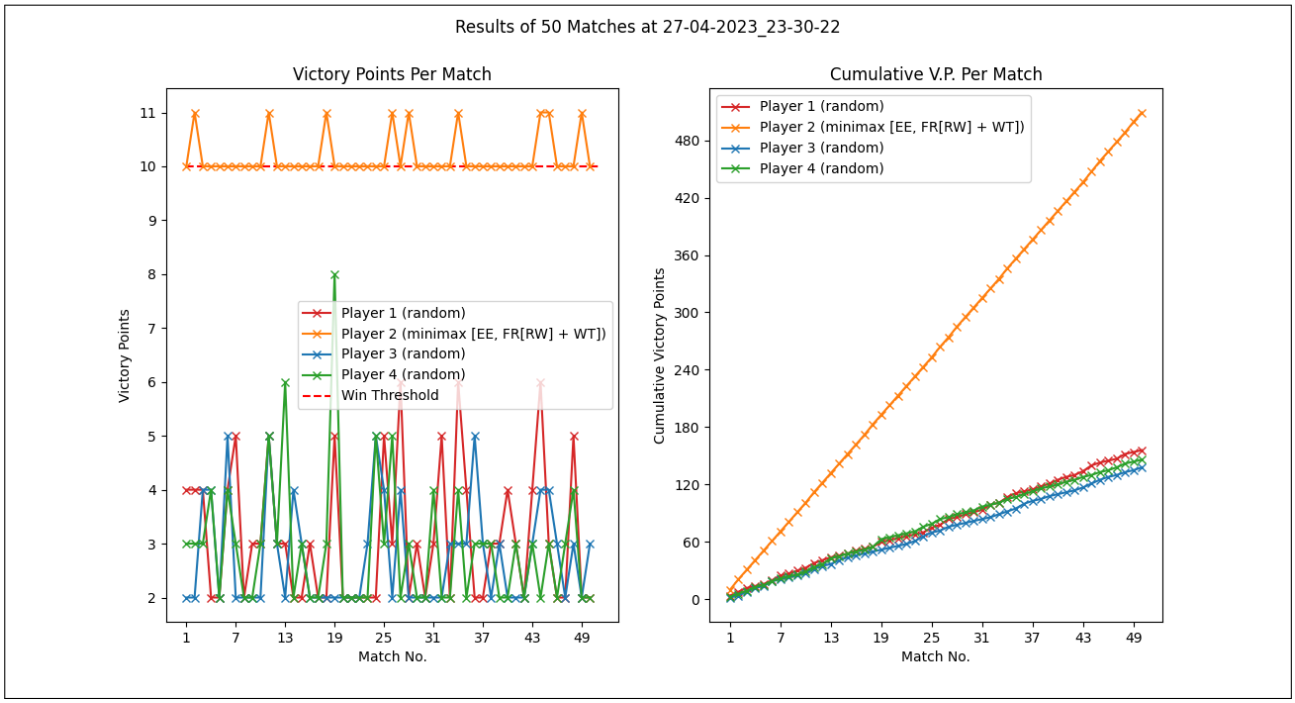


**Figure 23:** Evaluation results of the Early Expansion and Favour Resources (Rock and Wheat) Strategy

#### 9.1.2 Evaluation of Early Expansion and Favour Resources (Rock and Wheat) Strategy

Building on the previous strategy, the second best performing strategy was a combination between Early Expansion and Favouring Rock and Wheat. This strategy, as previously suggested, gives structure to both the start and end of the game. At the start, the player expands quickly to claim a large area, and during the mid and late game it builds cities using the Rock and Wheat tiles it is on. This strategy performs close to the best, winning 100% of its games with an average turns-to-win of 37.520. Interestingly, this strategy had the highest average victory points out of all strategies tested, however this makes sense after seeing in Fig. 23 the number of times the player had 9 VPs, and then gained 2 from the Largest Army of Longest Road.

#### 9.1.3 Evaluation of Development Card Strategy

As shown in Table 17, this strategy performs the best out of all. It wins 100% of test matches, and does so in an average turn count of 34.6, which is much higher than any other strategy tested previously. The author can identify two reasons why this strategy performs well. The first is that development cards are useful in claiming the Longest Road and Largest Army, which the MiniMax player likes to aim for. Soldier cards are the only method of getting the Largest Army, and road building cards, or even year of plenty cards help to build roads. The second reason is that the heuristic favours rock, sheep and wheat in order to buy more cards. Due to the fact that a city requires rock and wheat to build, a player using this strategy is in a much better position to try and build settlements, which are crucial later game. Therefore, these two factors enable the player to perform extremely well, in all stages of the game.

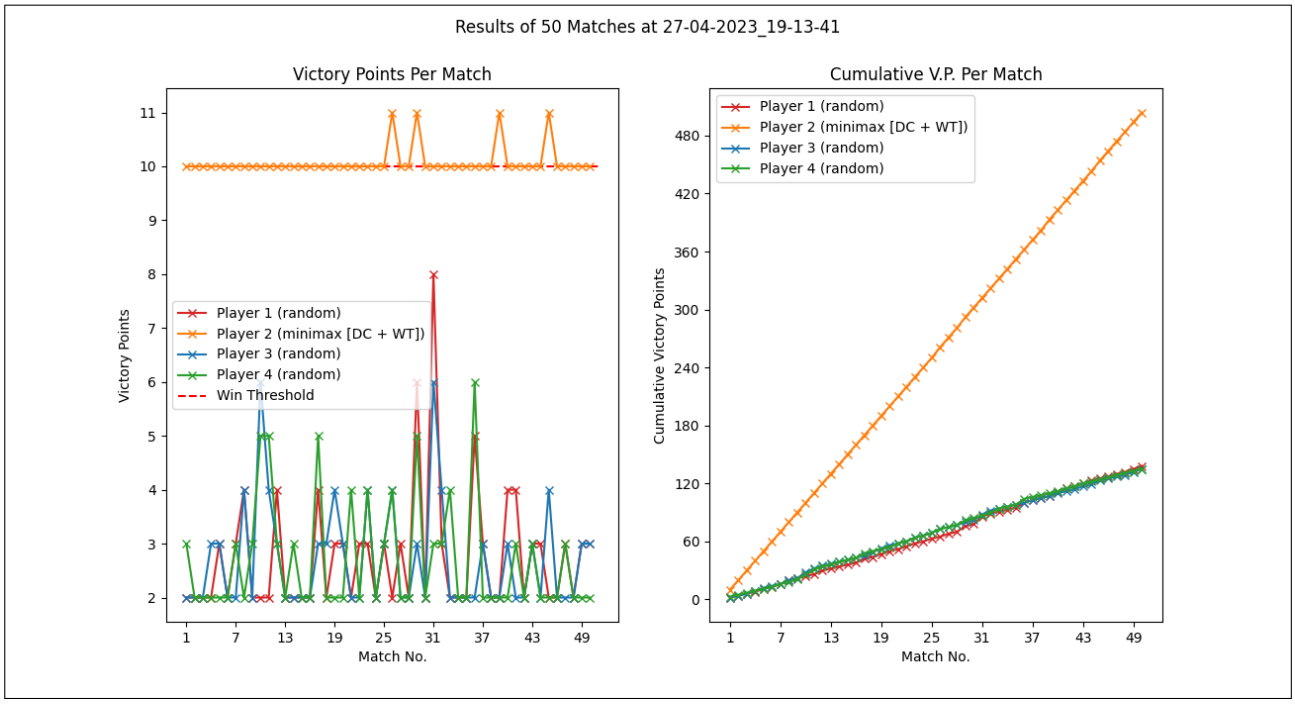


**Figure 24:** Evaluation results of the Development Card Strategy

### 9.2 Human Testing

Human testing of the program was conducted towards the end of the project, and was conducted in two phases. The first involved the author's family attempting to play a game with the program, with the author playing using an AI MiniMax player. A computer was set up next to the board, and the program was set up in tabletop mode (Fig. 25). During play, the author would input the moves to the program to keep it in sync with the real board, and would note down any UX issues or limitations of the AI. The full list of issues, across all rounds of testing, is included below.

- During setup, if the AI places two settlements one after the other, it is not clear which order they are placed in (Added)
- When a human is trading with another player, the players to trade with are shown in a numbered list, but the numbers don't match the player number. The list should not be numbered, to avoid confusion. (Fixed)
- The program does not say which development card was given to the AI, and does not say which card was stolen either by a human or an AI. (Fixed)
- A human can play a soldier card even if it has already been played before. (Fixed)
- It would be nice for human players to be given nicknames when playing in tabletop mode, to allow easier identification of players. (Added)
- Choosing to build a settlement or city as a human player would crash the game (Fixed)
- Placing a city would be refused as the program claimed that someone was already in that location (Fixed)

**Figure 25:** Human Testing Image taken by the author

- Players could not cancel actions such as building a road or settlement (Added)
- Denied trades would not be announced (Fixed)
- A MiniMax player could offer trades that it had already offered, if it randomly picked the same best move again
- The MiniMax player's trades often seemed to have no strategy. It would freely give away the clay that it had, after refusing a trade for it earlier.

While the aim of this phase was to test out both the User Experience and the complexity of the AI, the main information gained from it was UX related. Due to the nature of having to type in every move, it was not possible to play a complete game in this format, and so the strategy of the MiniMax player was not able to be fully tested. Instead, these rounds of testing were used for heavy testing of the human interaction side of the program, which is the other key area.

The second phase of testing included the author playing games with the MiniMax AI player using the best strategy discovered so far. The author was able to play two full games alongside many partial games with the MiniMax AI using the Development Card strategy. The graphs of the two full games are included below in Fig. 26. As can be seen, the MiniMax player performs well, often matching or exceeding the human player for a good portion of the game. In the first game, the MiniMax player had a good lead, but then lost the longest road and therefore dropped in points. In the second game, the AI performed well, but was fenced in, and so wasn't able to compete for the longest road. It also failed to build cities towards the end of the game, when it should do. Given more ideal situations, the author believes that the MiniMax player would be able to beat

them, and therefore they believe this AI is capable of being human competitive.

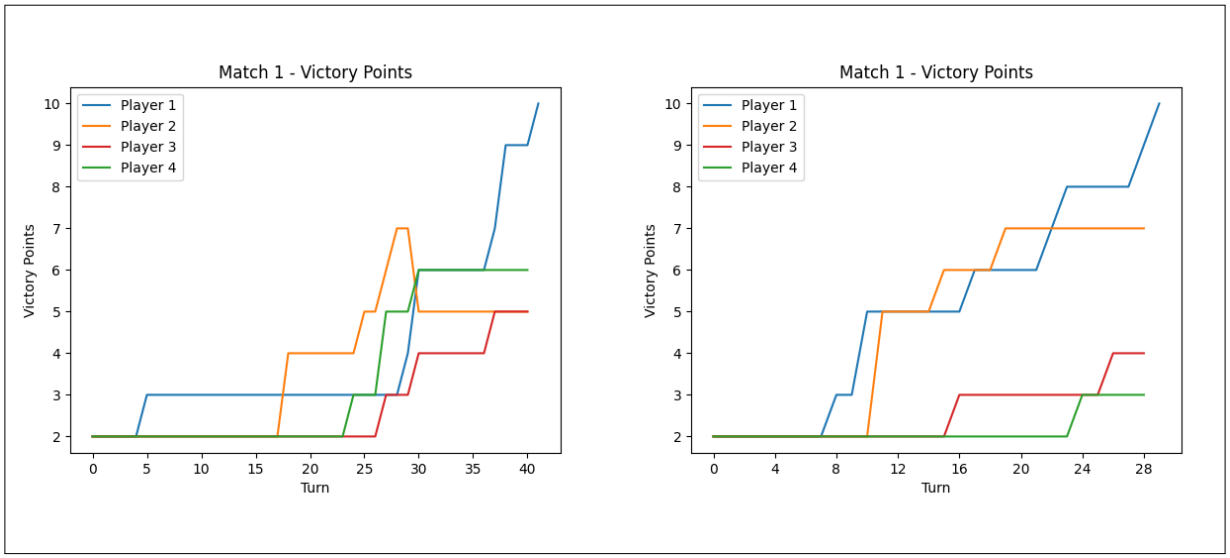


**Figure 26:** Graphs from two full games played between a human (Player 1), a MiniMax player (Player 2), and two random players (Players 3 and 4)

### 9.3 Observation on Behaviour

The author wishes to mention some general observations that they have observed when the MiniMax player is not given any strategy (other than Wishful Thinking), and is instead just left to decide on its own strategy.

- Longest Road and Largest Army
  - The MiniMax player often sees the Longest Road and Largest Army as 2 fairly easy Victory Points, as they can often be achieved in less than 10 turns if the right dice rolls are made. Therefore, the MiniMax player almost always goes for these, usually it goes for the Longest Road first, and then the Largest Army.
- Wheat and Rock
  - If there is no easy access to rock and wheat, the MiniMax player typically doesn't go for cities. This is because (assuming the player has either no rock or wheat), it would take a minimum of 2 successful trades to achieve enough to build a city. However, in that time, the player would normally be able to perform some other move that it believes would aid it better, such as building a road to reach a new location, or secure the longest road. Therefore, if the player does not have settlements on rock and wheat, it does not build cities.
- As can be seen in Fig. 27, the MiniMax player often has 1 or 2 long periods during a game where it does not gain any Victory Points.

– The author does not know the reason for this, but as the pauses often happen around 7-8 Victory Points, they speculate that it is because the MiniMax player has built as many settlements as it can, and is now searching to try and get the longest road or largest army. As mentioned previously, the MiniMax player does not build cities if it needs to trade for them. This limits the methods it can get VPs to only settlements, development cards, and the longest road or largest army. Provided the player has built all it's settlements, and has obtained either the longest road or largest army, it would have 7 victory points. This matches with the pauses often seen in the MiniMax player's behaviour.

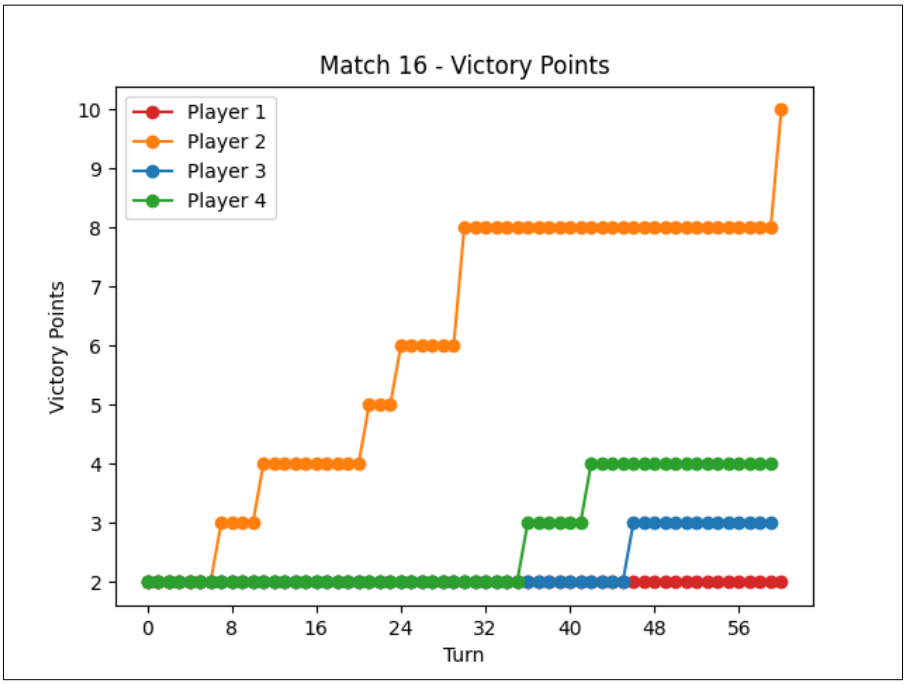


**Figure 27:** Victory Points vs turn graph of 10 matches played between a MiniMax player and 3 random players. MiniMax player is Player 2, shown in orange

# 10 Conclusions

Overall, the author believes that the project has been a great success. The AI player has performed extremely well using multiple different strategies, and can play at, or near to, a human competitive level, depending on the situation. There is also large potential for future research to be conducted, and the framework for this is already in place. The author has a personal connection with the project, and while it may not be perfect, optimal or complete, they hope that they will be able to use it in future games with their family to vary the gameplay, and see if it can finally help them win a game.

The author is most proud of their Heuristic Modifier system, as they believe that its modular nature, and ease of adding new statistics and scores, really lends itself to research. They are also proud of their extension of the MiniMax system, and while it may not be the most optimistic, they feel it still performs quite well, and makes up for its potential shortcomings in other ways. Finally, and mostly simply, they are proud of their recreation of Settlers. The variations from the original game are minor, and the game feels intuitive to play for anyone who has played Settlers before. Even without creating the AI, the project would be impressive, as converting a board game with elements such as trading, chance and random board layouts is no small feat.

The main limiting factors for the author has been time and computation power, as with an increase in both of these, the author may have been able to produce an AI that is even stronger than what has currently been developed. The author would have liked to have played more games with the AI, to understand more about its performance, and use these games to improve it. The author also wishes to perform more human testing in future to complete a full game with multiple people, to receive both algorithmic and UX feedback. Finally, they also wish to explore the future developments listed Section 10.2, as they believe these will improve the program for both researches and users.

## 10.1 Project Management

### 10.1.1 GitLab

As mentioned in Section 3.1, GitLab was used as the version control remote for the project. Commits were made on a fairly regular basis, often weekly, as can be seen in Fig. 28. GitLab often acted as a backup before making potentially breaking changes too, for example before adding parallelization to the program, the author made a commit to ensure progress was not lost.

### 10.1.2 JIRA

JIRA was used to keep track of tasks and issues for development, as well as bugs that needed fixing, as discussed in Section 8. A Kanban style board [32] was used, and updated whenever tasks were created, started or finished. The board also contained notes from the author's weekly meeting with their supervisor, and an 'epic' issue containing the Gantt Chart [33], included in Fig. 31. The issues in GitLab were assigned to stories to keep

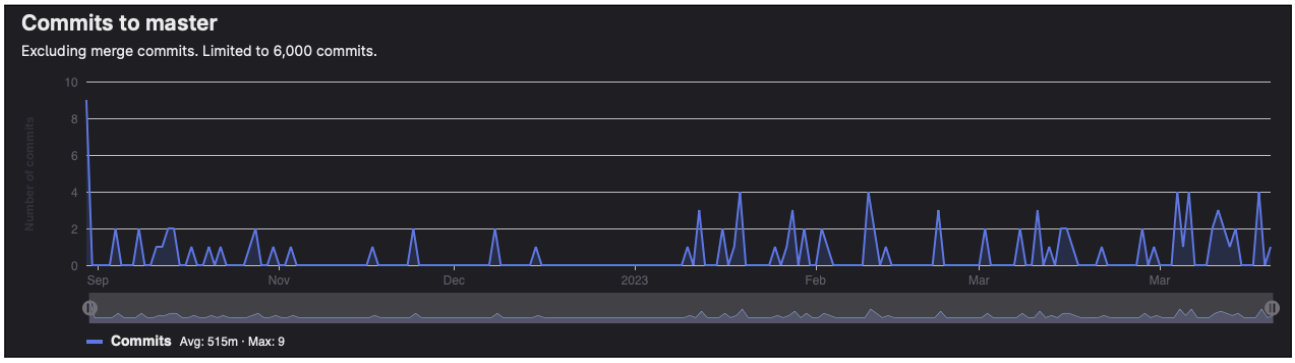


**Figure 28:** Commits to Gitlab, September 2022 - Aug 2023

them more organised, an examples of which could be 'Create MiniMax Player' with the subtasks of 'Create Board Heuristic', 'Create MiniMax Player Class', 'Investigate Max N Functionality' and 'Modify Heuristic'.

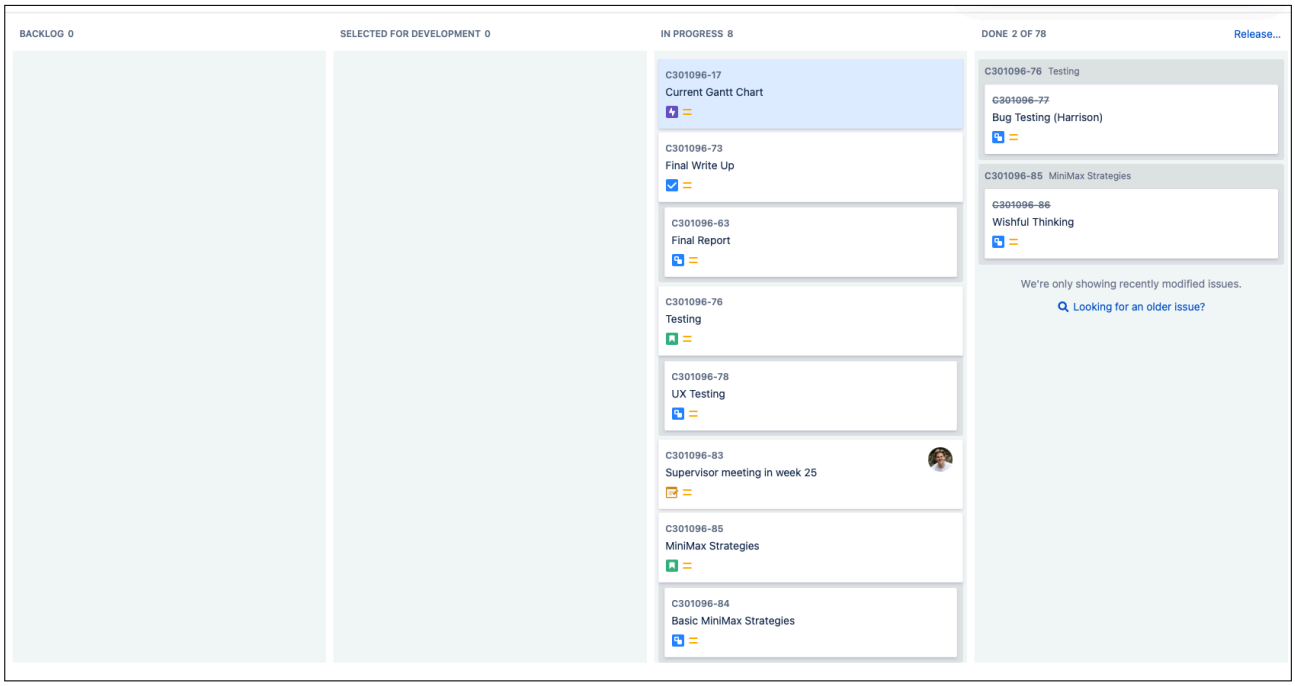


**Figure 29:** Screenshot of JIRA Kanban board, as of 20 April 2023

### 10.2 Further Developments

Due to the constrained-timeframe nature of the project, alongside the authors other commitments, there are multiple features and optimisations that the author would have liked to include but has not had time to. These are described below, alongside any thoughts the author has given to their implementation.

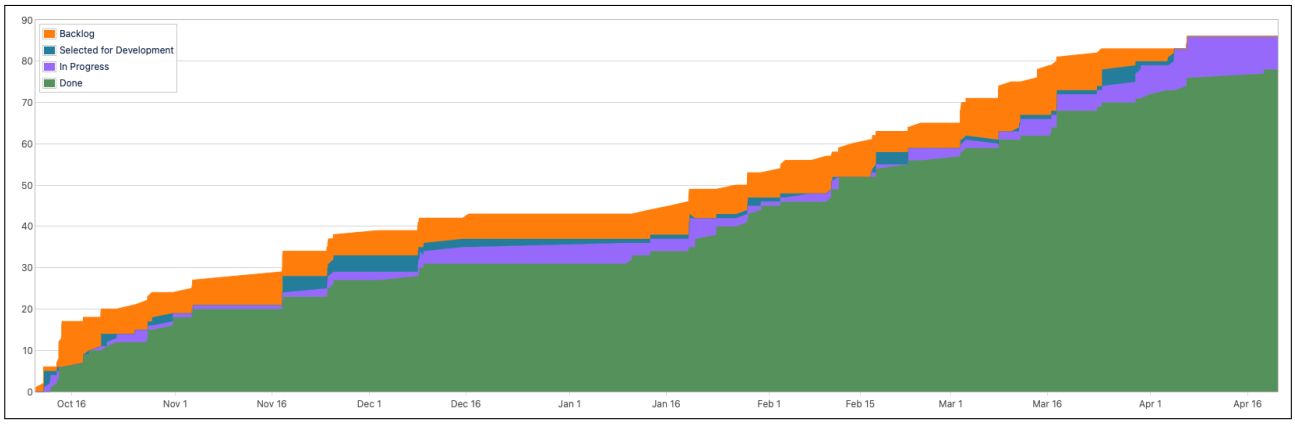


**Figure 30:** Cumulative Flow Diagram from JIRA, as of 20 April 2023

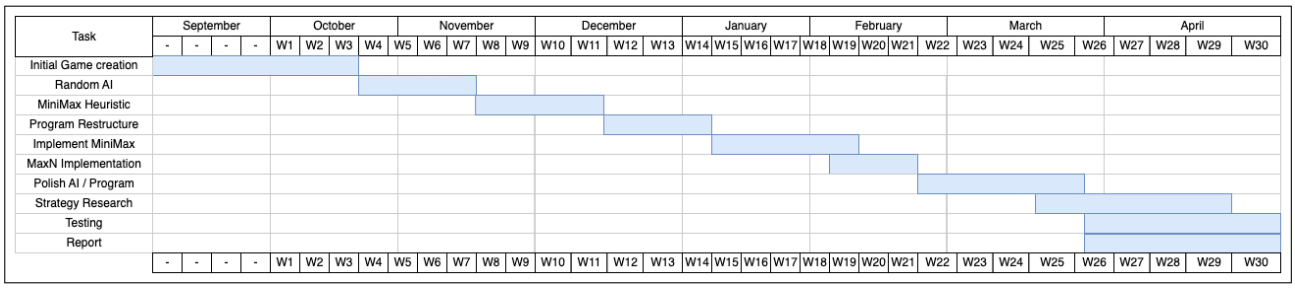


**Figure 31:** Gantt Chart, as of 25 April 2023

### 10.2.1 Board Improvements

**10.2.1.1 Misaligned Emojis Bug** There is one notable issue with the current display of the board, which is that occasionally when using a random layout for the board, the characters on a line are shifted one or two spaces too far to the left or right. This happens as the Apple Color Emojis used for the symbols have an indeterminate width when printed on the same line as other symbols, and the wrong combination can cause a blank space to appear or disappear. The author did try to correct for this by padding each symbols, but the issue is only present when the emojis are together, and so they were unable to rectify this within the timeframe of the project.

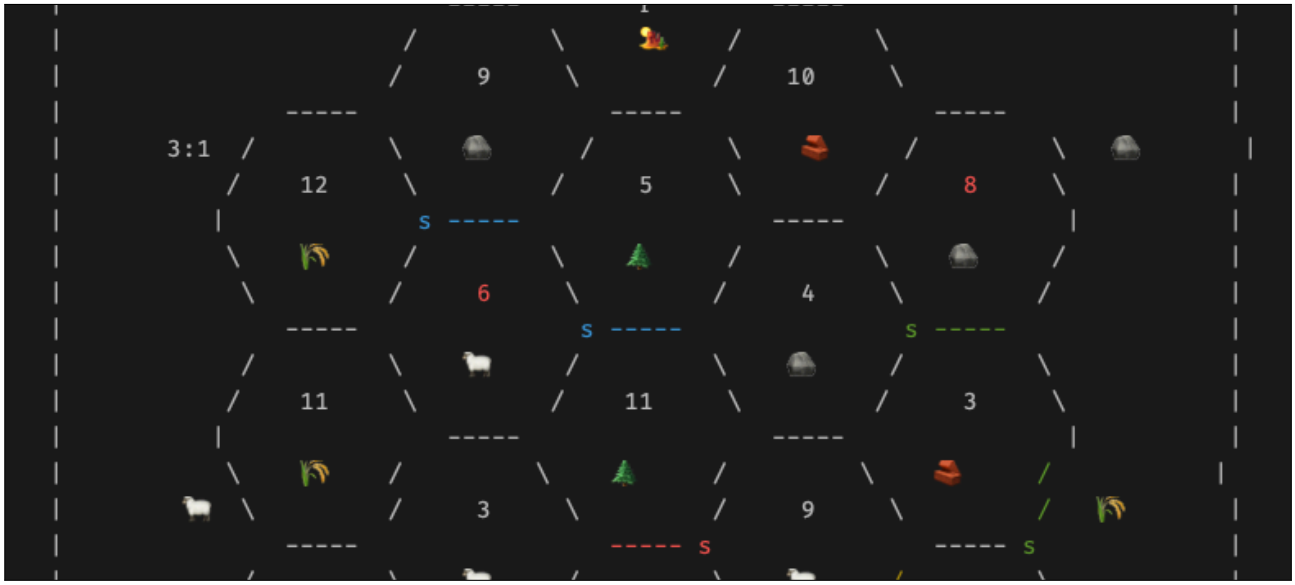


**Figure 32:** Section of a random board containing misaligned emojis

**10.2.1.2 Interaction** One current limitation identified by the author's supervisor is the interaction system for placing settlements, identified in Section 4.3. It is felt that the grid reference style approach does not feel native, and would be much better replaced by a mouse or drag-and-drop system, or even an arrow keys system. The author agrees that this system is most likely more finicky than it needs to be. However, changing to arrow keys, or upgrading from a keyboard based system entirely would involve designing a whole GUI, as a mouse or drag-and-drop system cannot be used with the command line in this regard. Therefore, the author decided that this was not worth the time that it would take, especially with a large focus of the project being on research as opposed to shipping a commercial product. However, given more time, the author would like to add a GUI system, as this would potentially allow to easier testing and data collection with human players.

**10.2.1.3 Custom Board Layouts** The ability to add a custom board layout is quite an easy feature, and would improve the experience of tabletop mode. The author believs that all this would require is a setup function that prompts the user repeatedly for a tile at given locations until the board is complete. The benefits of adding

this would be that when playing with two boards, the program no longer needs to be the master copy of the board, and instead the physical copy can be the master.

### 10.2.2 MiniMax Improvements

**10.2.2.1 MiniMiniMax** As detailed in Section 6.4, the current implementation of MiniMax for more than 2 players simply extends MiniMax to become MiniMiniMax, where every other play is assumed to make the worst possible move in regard to the MiniMax player. The author believes that this can be improved upon, in multiplayer games like this it is unfair to assume that every other opponent will 'gang up' on the MiniMax player, and therefore the MiniMax player's overly defensive or conservative play-style is not necessary. The author feels that a more balanced approach of sometimes assuming the worst (minimising), or sometimes not (going for a middle of the road option), could be more appropriate. This could be known as a Mini-Mid-Max approach. The level of 'Mini' or 'Mid' could be based on the position of the other player, for example a strong opponent may well need to be treated as a 'Mini', but a weak opponent is more likely to be focused on themselves, and so a 'Mid' approach could be taken.

### 10.2.3 MaxN

Currently, the player's extension of MiniMax to move than 2 players can result in somewhat pessimistic planning. While the author identified that this is not a large issue within Settlers, they would like to implement the aforementioned MaxN algorithm. As mentioned earlier, MaxN is an extension of MiniMax that provides improvements over MiniMiniMax by assuming each player will make the move best for them. This is based off of the assumption that all the other player's won't team up on the MiniMax player, which is more reasonable. The author did attempt to add this, but was unable to modify their pre-existing MiniMax code, and therefore a full rewrite of the MiniMax algorithm would be required to add this.

### 10.2.4 Hyperparameter Tuner

The heuristic defined in Section 6.2 contains a number of constant values for scoring various items, such as +50 if the player is tied for winning, or -5 * the number of players in the road. These values were chosen by the author based on observations of the game, however, these may not be the optimum values. As there are so many values, the author would like to utilise a Hyperparameter Tuner, such as KerasTuner [34], which can be used to find the best values across the entire heuristic. This is done using an optimisation function, aiming to increase the players win rate or decrease the player's turns-to-win score. However, this would require many calculations and games run in order to test out many combinations, and so this may not be computationally possible.

#### 10.2.5 Genetic Algorithm

Due to the modular nature of the heuristic modifiers, the author theorises that it would be possible to use a Genetic Algorithm [35] to find the best combination of heuristic modifications. Genetic Algorithms are algorithms designed to solve combinatorial problems through methods of natural selection, by evolving potential solutions towards a goal. The process starts by creating a population of potential candidates, in this case these would be combinations of heuristic. Then, these would all be evaluated, and the best ones would be bred to produce similar candidates which replace the worst performing ones from the last generation. This process is repeated either for a set time, or until a score is met. The author theorises that this *would* be able to find a promising combination of heuristic modifications, but (much like the previous point) the time to do so would be far too high. They feel that it would take at least 10 simulated games in order to fairly evaluate each combination, and performing this amount for any decent sized population would just be computationally impossible. In order for this to be feasible, some form of large optimisation method would need to be implemented, either for the Genetic Algorithm, or for the speed of the game.

#### 10.2.6 Monte–Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is another heuristic search algorithm, similar to MiniMax in that it solves a game tree, but different in its process. MCTS does not require a heuristic, and instead functions by selecting a branch, and performing 'playouts', where a branch of moves is played out until the end of the game, and the result noted. This result is then propagated back up the branch, increasing the branches score if it resulted in a win, otherwise decreasing it. The main tuning method for this comes from the selection method, as the algorithm can be favoured towards selecting nodes that haven't been explored much, and so could be promising (exploration), or choosing nodes that have been the most promising so far out of all the nodes that have been explored (exploitation). The author would like to add MCTS as its own AI in the future, to determine whether this self-taught player would be able to beat the MiniMax player with its hand designed heuristic.

## References


[1] I. Zuckerman, B. Wilson, and D. S. Nau, "Avoiding game-tree pathology in 2-player adversarial search: Avoiding game-tree pathology in search," *Computational Intelligence*, vol. 34, no. 2, pp. 542–561, May 2018, doi: 10.1111/coin.12162. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/coin.12162. [Accessed: Apr. 25, 2023]

[2] B. Mahoney, "How to play: Settlers of catan," *Medium*. [Online]. Available: https://medium.com/board-game-brother/how-to-play-settlers-of-catan-e51c5a5aa499. [Accessed: Mar. 28, 2023]

[3] "Game rules | CATAN." [Online]. Available: https://www.catan.com/understand-catan/game-rules. [Accessed: Mar. 28, 2023]

[4] Stuart Russel and Peter Norvig, *Artificial Intelligence: A Modern Approach, 4th US ed.* [Online]. Available: http://aima.cs.berkeley.edu/. [Accessed: Apr. 24, 2023]

[5] N. N. (Nmnogueira), "MiniMax example image," *Wikipedia*. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/6/6f/Minimax.svg. [Accessed: Apr. 18, 2023]

[6] N. Metropolis and S. Ulam, "The Monte Carlo method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, Sep. 1949, doi: 10.1080/01621459.1949.10483310.

[7] "Heuristics - Definition and examples," *Conceptually* [Online]. Available: https://conceptually.org/concepts/heuristics. [Accessed: Apr. 25, 2023]

[8] D. J. Edwards and T. P. Hart, "The Alpha-Beta Heuristic," Dec. 1961 [Online]. Available: https://dspace.mit.edu/handle/1721.1/6098. [Accessed: Apr. 24, 2023]

[9] *MacUser January 1986*. 1986 [Online]. Available: http://archive.org/details/MacUser8601January1986. [Accessed: Apr. 18, 2023]

[10] Jez9999, "MiniMax example image," *Wikipedia*. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/6/6f/Minimax.svg. [Accessed: Apr. 18, 2023]

[11] I. Szita, G. Chaslot, and P. Spronck, "Monte-Carlo Tree Search in Settlers of Catan," in *Advances in Computer Games*, 2010, pp. 21–32, doi: 10.1007/978-3-642-12993-3_3.

[12] "Monte carlo tree search in a modern board game framework," pp. 1–11, 2012 [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/bsc/Roelofs_Bsc-paper.pdf. [Accessed: Apr. 26, 2023]

[13] L. Fox, *Advances in programming and non-numerical computation: Symposium 1963*, 1. ed. Oxford: Pergamon Pr, 1966.

[14] J. Austin and S. Molitoris-Miller, "The Settlers of Catan: Using Settlement Placement Strategies in the Probability Classroom," *The College Mathematics Journal*, vol. 46, no. 4, pp. 275–282, 2015, doi: 10.4169/college.math.j.46.4.275. [Online]. Available: https://www.jstor.org/stable/10.4169/college.math.j.46.4.275. [Accessed: Apr. 26, 2023]

[15] M. Guhe and A. Lascarides, "Game strategies for The Settlers of Catan," *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pp. 1–8, Aug. 2014, doi: 10.1109/CIG.2014.6932884. [Online]. Available: https://www.research.ed.ac.uk/en/publications/game-strategies-for-the-settlers-of-catan. [Accessed: Apr. 26, 2023]

[16] "Welcome to Python.org," *Python.org*. Apr. 2023 [Online]. Available: https://www.python.org/. [Accessed: Apr. 21, 2023]

[17] JetBrains, "PyCharm: The Python IDE for Professional Developers by JetBrains." [Online]. Available: https://www.jetbrains.com/pycharm/. [Accessed: Mar. 29, 2023]

[18] Atlassian, "Sourcetree | Free Git GUI for Mac and Windows." [Online]. Available: https://www.sourcetreeapp.com. [Accessed: Mar. 29, 2023]

[19] Atlassian, "Jira." [Online]. Available: https://jira.atlassian.com/. [Accessed: Mar. 29, 2023]

[20] A. Sottile, "Pre-commit." [Online]. Available: https://pre-commit.com/. [Accessed: Mar. 29, 2023]

[21] Ł. Langa, "Black: The uncompromising code formatter." [Online]. Available: https://github.com/psf/black. [Accessed: Mar. 29, 2023]

[22] "Apple Color Emoji," *Wikipedia*. Mar. 2023 [Online]. Available: https://en.wikipedia.org/w/index.php?title=Apple_Color_Emoji&oldid=1145412464. [Accessed: Mar. 28, 2023]

[23] "Pydeps: Display module dependencies." [Online]. Available: https://github.com/thebjorn/pydeps. [Accessed: Mar. 28, 2023]

[24] "Algorithms Explained – minimax and alpha-beta pruning." [Online]. Available: https://www.youtube.com/watch?v=l-hh51ncgDI. [Accessed: Apr. 19, 2023]

[25] "Serialization," *Wikipedia*. Jan. 2023 [Online]. Available: https://en.wikipedia.org/w/index.php?title=Serialization&oldid=1132012847. [Accessed: Apr. 17, 2023]

[26] "Ujson: Ultra fast JSON encoder and decoder for Python." [Online]. Available: https://github.com/ultrajson/ultrajson. [Accessed: Apr. 18, 2023]

[27] "Jsonpickle: Python library for serializing any arbitrary object graph into JSON." [Online]. Available: https://github.com/jsonpickle/jsonpickle. [Accessed: Apr. 18, 2023]

[28] Electro, "Deepcopy() is extremely slow," *Stack Overflow*. Nov. 2017 [Online]. Available: https://stackoverflow.com/q/24756712. [Accessed: Apr. 17, 2023]

[29] "Concurrent.futures — Launching parallel tasks," *Python documentation*. [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html. [Accessed: Apr. 17, 2023]

[30] Prof.Chaos, "Answer to "Extending minimax algorithm for multiple opponents"," *Stack Overflow*. Aug. 2020 [Online]. Available: https://stackoverflow.com/a/63609301. [Accessed: Apr. 24, 2023]

[31] *An Algorithmic Solution of N-Person Games*. [Online]. Available: https://aaai.org/papers/00158-an-algorithmic-solution-of-n-person-games/. [Accessed: Apr. 25, 2023]

[32] Atlassian, "Kanban - A brief introduction," *Atlassian*. [Online]. Available: https://www.atlassian.com/agile/kanban. [Accessed: Apr. 26, 2023]

[33] Atlassian, "Gantt Charts," *Atlassian*. [Online]. Available: https://www.atlassian.com/agile/project-management/gantt-chart. [Accessed: Apr. 26, 2023]

[34] K. Team, "Keras documentation: Getting started with KerasTuner." [Online]. Available: https://keras.io/guides/keras_tuner/getting_started/. [Accessed: Apr. 26, 2023]

[35] M. Mitchell, *An introduction to genetic algorithms*. Cambridge, Mass.: MIT Press, 1996.

## Appendix A. Tables

**Table 7:** Tile Frequencies based on Dice Roll

| *Tile Number* | *Frequency* |
|---|---|
| 6 or 8 | 5 |
| 5 or 9 | 4 |
| 4 or 10 | 3 |
| 3 or 11 | 2 |
| 2 or 12 | 1 |

### Game Testing Results

**Table 8:** 10 Games played between a MiniMax player using the Favour Resources (Clay and Wood) and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [FR[CW] + WT]) | 70% | 9.3 | 98.936 | 6.68s |
| Player 4 (random) | 30% | 5.5 | 166.683 | 0.03s |
| Player 1 (random) | 0% | 5 | 177.271 | 0.03s |
| Player 3 (random) | 0% | 4.8 | 172.539 | 0.03s |

**Table 9:** 10 Games played between a MiniMax player using the Favour Resources (Rock and Wheat) and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [FR[RW] + WT]) | 90% | 9.6 | 69.508 | 8.56s |
| Player 4 (random) | 10% | 5.2 | 133.088 | 0.03s |

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 1 (random) | 0% | 3.4 | 202.333 | 0.03s |
| Player 3 (random) | 0% | 3.7 | 182.2 | 0.03s |

**Table 10:** 10 Games played between a MiniMax player using the 'Development Card Spam' and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [DC + WT]) | 100% | 10.2 | 55.3 | 3.54s |
| Player 1 (random) | 0% | 4.4 | 132.778 | 0.03s |
| Player 3 (random) | 0% | 3.7 | 147.333 | 0.03s |
| Player 4 (random) | 0% | 4 | 168.567 | 0.03s |

**Table 11:** 10 Games played between a MiniMax player using the Early Expansion and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [EE + WT]) | 100% | 10.2 | 59.6 | 4.23s |
| Player 1 (random) | 0% | 3.6 | 180.181 | 0.03s |
| Player 3 (random) | 0% | 2.8 | 215.417 | 0.03s |
| Player 4 (random) | 0% | 3.5 | 178.464 | 0.03s |

**Table 12:** 10 Games played between a MiniMax player using the No Ports and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [NP + WT]) | 70% | 9.1 | 95.451 | 6.05s |
| Player 1 (random) | 20% | 5.1 | 176.583 | 0.03s |
| Player 3 (random) | 10% | 5.4 | 178.413 | 0.03s |

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 4 (random) | 0% | 4.4 | 187.933 | 0.03s |

**Table 13:** 10 Games played between a MiniMax player using the Early Expansion, Favour Resources (Wood and Clay) and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [EE, FR[WC] + WT]) | 80% | 9.6 | 91.8 | 7.41s |
| Player 1 (random) | 10% | 4 | 210.861 | 0.03s |
| Player 4 (random) | 10% | 5.1 | 153.205 | 0.03s |
| Player 3 (random) | 0% | 4 | 191.667 | 0.03s |

**Table 14:** 10 Games played between a MiniMax player using the Early Expansion, Favour Resources (Rock and Wheat) and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [EE, FR[RW] + WT]) | 90% | 10.1 | 59.078 | 6.49s |
| Player 4 (random) | 10% | 4.2 | 139.99 | 0.03s |
| Player 1 (random) | 0% | 3 | 197.9 | 0.03s |
| Player 3 (random) | 0% | 3.7 | 163.188 | 0.03s |

**Table 15:** 50 Games played between a MiniMax player using the Early Expansion and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [EE + WT]) | 100% | 10.02 | 44.544 | 3.42s |

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 1 (random) | 0% | 2.88 | 155.423 | 0.02s |
| Player 3 (random) | 0% | 3.18 | 140.060 | 0.02s |
| Player 4 (random) | 0% | 3.1 | 146.383 | 0.01s |

**Table 16:** 50 Games played between a MiniMax player using the Early Expansion. Favour Resources (Rock and Wheat) and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [EE, FR[RW] + WT]) | 100% | 10.18 | 37.520 | 2.46s |
| Player 1 (random) | 0% | 3.12 | 130.490 | 0.05s |
| Player 3 (random) | 0% | 2.76 | 144.563 | 0.04s |
| Player 4 (random) | 0% | 2.92 | 133.338 | 0.05s |

**Table 17:** 50 Games played between a MiniMax player using the 'Development Card Spam' and Wishful Thinking Modifiers

| *Player* | *Win Rate* | Avg Victory Points | Avg Turns to Win | Avg Turn Time |
|---|---|---|---|---|
| Player 2 (minimax [DC + WT]) | 100% | 10.08 | 34.640 | 1.95s |
| Player 1 (random) | 0% | 2.76 | 133.835 | 0.29s |
| Player 3 (random) | 0% | 2.70 | 134.417 | 0.19s |
| Player 4 (random) | 0% | 2.70 | 129.193 | 0.33s |