Third Year Dissertation Project - G400

University of Nottingham

School of Computer Science

Evolutionary Development of Ancient Board Game Heuristics

Dylan Devalia

4262344 (PSYDD2)

supervised by

Venanzio Capretta

I hereby declare that this dissertation is all my own work,
except as indicated in the text

Signature: ______________________________________

24th April 2018

Abstract

This dissertation will present a digital version of *The Royal Game of Ur*, one of the oldest known board games. The paper consists of building the game from scratch then creating an artificial intelligent opponent using machine learning techniques, namely evolutionary genetic algorithms. The aim of this project is to try and bring the new life into the board game by developing a clean, accessible version of the game which anybody can play, as well as creating an AI which can play the game to a high standard for the public to play against. To test the strength of the AI, participants were asked to play several agents with different behaviours to identify which AI was the most fun to play against.

The paper consists of two sections: creating the game engine in *Java* including the game loop, state manager, and canvas rendering; and developing the AI by studying the different strategies the game can be played and applying them to a genetic algorithm. Participant will then play three version of the AI, one using aggressive behaviour, a bespoke designed agent, and the evolved agent.

Keywords: *board game, game engine, genetic algorithm, machine learning*

Acknowledgements

I would like to start off by giving my earnest thanks to my supervisor, Venanzio Capretta, for allowing me to pursue this study, as well as giving his guidance, assistance, and encouragement throughout the project.

I am also indebted to all the friends and family who have supported me from day one and their constructive suggestions that helped shaped the final project.

Lastly, I would give a special gratitude to the University of Nottingham and its School of Computer Science. I have learnt a great deal throughout the three years studying under the fantastic academics.

Contents

A Brief History

In 1925, archaeologists working in Iraq and eastern Syria, sponsored by the University of Pennsylvania and the British Museum, discovered tombs near the ancient Sumerian city of Ur. Enclosed in the mausoleum, along with royal treasures and religious artifacts, several board games were discovered dating back as far as 3000B.C.E [1].

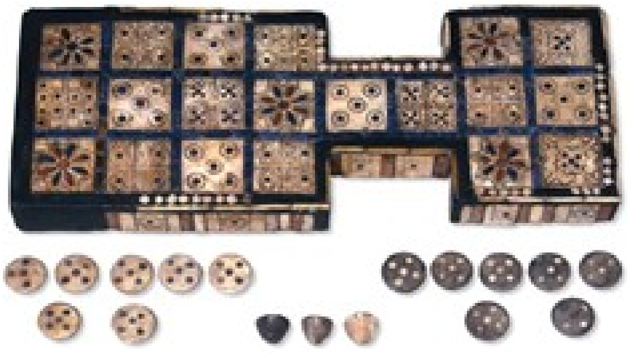


Figure 1: The Royal Game of Ur game board, dice, and counters [2]

One of the oldest of the games found became known as *The Royal Game of Ur*. Although the original rules have been lost to time, researchers have been able to reconstruct the game using clues such as the design of the board, counters and dice alongside tablets from around 200B.C.E. These newly discovered tablets had rules etched into them for a very similar game, although from a much later era [2].

Chapter 1

Background and Motivation

This chapter will introduce the project and the decisions behind choosing the topic, as well as the aims and objectives hoped to be achieved in the final solution. Additionally, the chapter will cover any preceding knowledge that may be required to understand the concepts discussed in the paper. Finally, any previous work which helped inform the basis and development of the project will be considered and examined.

1.1 Introduction

The objective of this project is to create and formidable artificial intelligence opponent for the ancient board game *The Royal Game of Ur.* The first step towards this goal is to create a digital version that can be played by anyone. For the project's purpose, the solution only requires a single user (player) since the opponent would be the AI, but the final solution should support two human players that can play against each other.

The artificial intelligence will be built from the ground up with a single-agent architecture that will evaluate all possible moves on every given turn and act according to its best interest. It will calculate its most optimal move by assessing various in-built strategies and choosing the most beneficial option depending on its current position.

Once the strategy system is in place, a genetic algorithm will be used to evolve several AIs by assigning weights to each technique and, though evolutionary algorithms, these weights should become fine-tuned to the values that create the best AI.

1.1.1 Aims

The aim of this project is to bring *The Royal Game of Ur* back into the public image by creating and developing an accessible, digital rendition of the ancient board game that can be played by anyone. The game will also feature an artificial intelligence opponent which can play to different strategies depending on its current position on the board. The AI will be trained using a machine learning algorithm which will help inform the agent which strategies are the most optimal.

1.1.2 Objectives

The key objectives of this project are:

1. To create an accessible digital adaptation of *The Royal Game of Ur*, playable by both human or AI players
2. Investigate existing artificial intelligence methods used for turn-based board games
3. Adapt researched techniques and implement into digital game
4. Explore the various strategies of *The Royal Game of Ur* and manually create instances of the AI to play to those approaches
5. Use genetic algorithm techniques to create an artificial opponent that has created its own play-style over generations of evolution

1.2 Motivation

In the recent years there has been an enormous shift in the field of computer science towards artificial intelligence and machine learning. One illustration of this is the work of internet super-giant *Google*[1]. Over the past decade, Google has made a massive push towards their deep learning research and development, now with the ability to correctly identify music though listening with microphones [4], a person's face from photos and pictures [5], adjusting map routes depending on real-time traffic analysis [6],

[1] Google is owned by their parent company Alphabet which also have subsidiaries such as DeepMind, Nest Labs and YouTube. Although all these companies use AI in diverse ways, this paper will refer to Google synonymously due to its global recognisability [3]

as well as recommending videos you may enjoy based on previously watched content [7].

In addition to improving their public services through machine learning, Google have also created several solutions to complex problems to test and showcase their machine learning capabilities. The most relevant examples are the AIs created to playing the complex board games *chess* and *go*, which have become successful enough to even beat grand-masters in their respective field [8].

It is in this area of study, which inspired the project for this thesis, allowing a machine to make intelligent choices and eventually be able to beat human opponents in strategic board games. Although Google's algorithms use neural networks [8], they evaluate hundreds of thousands of games to analyse past data; a task which the time limitations on the project would not allow. Evolutionary development is significantly cheaper and time-efficient to develop and, although it cannot offer the same sophistication, can produce comparable results [9].

After choosing an approach, a game was needed to tackle. Several mathematical games were considered, such as *Alquerque* which is a grid-based board game with counters, originally found in the Middle East [1]; and *Rithmomachy*, an early European board game similar to chess except capturing pieces depends on numbers etched onto the counters [1]. Ultimately, *The Royal Game of Ur* was chosen for its rich interesting history and intriguing chance-strategy hybrid mechanics.

1.2.1 Background Knowledge

This section will present all required knowledge that is relevant to the project. The first part will describe the rules of *The Royal Game of Ur* and instructions how to play a match. The following section biefly go over the history of artifical intelligence and the ideas that Alan Turing presented. Finally, the last section will walk through a genetic algorithm step-by-step [10] and explore the pros and cons of the machine learning technique [11].

How to Play *The Royal Game of Ur*

The objective of this two player, turn-based competitive game is to move all 6 of the player's counters around the board (board shown in *figure 1*) before their opponent does the same with their counters.

The game uses specialised dice shaped as pyramids (tetrahedrons) with white pips in two of the four corners. Once thrown, the number of pips facing upwards is that player's score (see *figure 1.2.* That score is then used to move a single counter by that many tiles. The first player (usually white counters) move around the board in the route shown in *figure 1.1.* The other player (black counters) moves in a similar route but mirrored along the middle row.

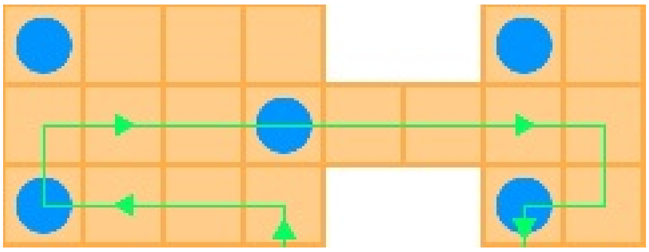


Figure 1.1: Simplified view of the board showing counter path

Since the middle row is shared between both parties, the counters are at war! If the player's counter lands on the opposing player's counter, the opposing player's counter is sent back to its starting position off the board.

Finally, if the player's counter lands on a rosette tile - marked as blue circles on *figure 1.1* - the player gains an extra turn. An additional benefit of landing on a rosette tile is any counter safe from being captured by an enemy piece [2].

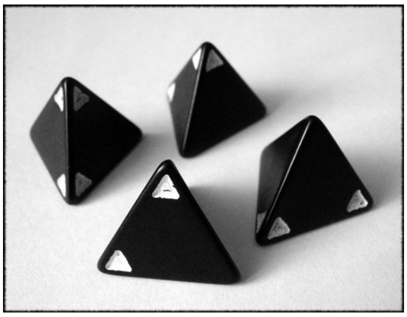


Figure 1.2: View of a modern interprtation of the dice used in *The Royal Game of Ur*. Current roll would have a score of 3 [12]

Artificial Intelligence and Turing Tests

Artificial intelligence is an ever-growing field of research in computer science since the early 20th century. Mathematician and early computer scientist *Alan M. Turing* first posed the question "Can machines think?" [13] in his 1950 paper *Computing Machinery and Intelligence* where he discusses the meanings of "machine" and "think" in modern culture and puts forth his *imitation game* [13], more commonly known as the Turing test [9].

The Turing test is a hypothetical thought experiment in which an interrogator interviews two subjects, a human and a machine. The subjects are hidden behind a screen so that when the interrogator asks a question they cannot tell which subject has answered. The test is successful when the interrogator cannot differentiate between the subjects [13].

In the modern age, our concept of artificial intelligence has expanded from mimicking human behaviour to giving machines the ability to learn through trial and error. This branch of AI is known as machine learning.

There are many techniques of teaching machines. Two of the most popular are genetic algorithms and neural networks [9]. Genetic, or evolutionary, algorithms train machines over several generations by starting with a group of randomly generated AI instances and slowing evolving them over time to become competent, and ultimately the best, at the given task.

Neural networks, on the other hand, develop neurons like how a brain functions. By being trained on past data the algorithm develops a structure of these neurons which can make decisions based on stimuli [14]. However, instead of evolving over time by performing the task over generations, neural networks process past data of the task and create an inter-connected system of logic.

1.3 Related Work

This section will outline, and detail other works of artificial intelligence being used to beat humans at games. The first section will describe how simple algorithms for strategic games work using the examples of *Noughts and Crosses*[2] and *Connect Four*. The following section will discuss of the two machines which beat human grand-masters in *Chess* and *Go*, called *Deep Blue* and *AlphaGo* respectively.

1.3.1 Noughts and Crosses & Connect Four

Noughts and Crosses is a simple game consisting of a 3x3 grid. The game is played by two players where player one is crosses ('X') and player two is noughts ('O'). Each turn the players places their token in an empty cell in the grid. The winner is the first player to complete a line of 3 of their tokens, either vertically, horizontally or diagonally [15].

Connect Four similarly uses a grid structure but it is usually played with a 3D model rather than with pen and paper. The grid is usually 7 wide and 6 tall although other size boards do exist. The object of the game is for both players to take turns dropping their respective counters from the top into a certain column. The counter will the proceed to fall until it either

[2] As known as *Tic Tac Toe* in counties such as the United States and Canada

reaches to the bottom of the grid or lands on top of another counter. Again, similarly to *Noughts and Crosses*, the aim of the game is for the players to achieve a consecutive line of their counters either vertically, horizontally or diagonally. As the name would suggest, in *Connect Four* players must match four counters to win [15].

There are a few ways to create algorithms for these games. One of which is known as '*minimaxing*'. Minimaxing is the approach of first creating a game tree or decision tree which is a network of all possible moves in the game [15]. In *Noughts and Crosses*, for example, the top of the tree would be the empty 3x3 grid. Underneath that would be nine different grids representing the nine unique moves a player could make. Underneath each of those grids would be eight other grids representing the eight moves the second player is able to make, and so on and so forth until a branch is won and thus there is no need to continue or the game ends in a draw when the board is full (see *figure 1.3*) [16].

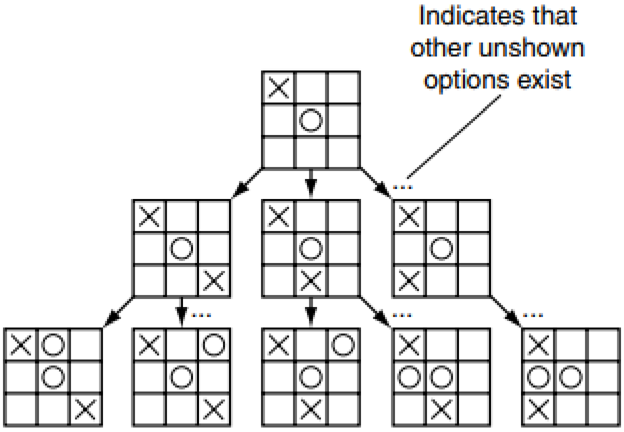


Figure 1.3: A partial decision tree for *Noughts and Crosses* [15]

Once the tree is completed, it is then analysed to find the strongest moves in each branch creating a tree which, ultimately, would be able to select the best possible move for any scenario. Unfortunately, this approach gets out of hand increasingly quickly. Even for a simple game such as *Noughts and Crosses*, there are already over 6000 different states to compute and store in memory[3]. Expanding this idea to *Connect Four* which has a seven possible moves a round (the number of columns in the board), by turn four there are already over 2000 possible states.

Instead of pre-computing all possible results it is more reasonable to

[3] Actually, *Noughts and Crosses* is an interesting example since the board can be rotated and flipped to save on states since the grid is symmetrical. By using this mathematical efficiency, the actual number of states can be simplified to just 96! [17]

have the algorithm check the available moves during the game. During the machine's turn, it would simulate and analyse potential actions, usually with a depth parameter informing the machine how many turns in advance to compute. The machine would still have to check a myriad of states in more complex games. A potential solution is the use of *AB Pruning* (or *Alpha-Beta Pruning*) [15].

AB Pruning allows the algorithm to actively ignore sections of the tree which could not possibly contain the best move [15]. It assigns values to the possible states; higher the value the stronger the move. Using these values, the pruning algorithm can ignore states that are have lower values that the current best.

1.3.2 Deep Blue & AlphaGo

Deep Blue is a world-renowned computer developed by IBM in the early 90s to play the board game *Chess* [18] and it was the first machine able to defeat a world champion. Garry Kasparov, born in Russia in 1963, became the first world-class challenger to be defeated by a computer under regular time controls. Kasparov and Deep Blue battled twice, once in 1996 and again in 1997, in a six-game match.

During their first bout, although Deep Blue managed to win the first game of six, Kasparov managed to be triumphant, winning 4-2. After its defeat, the IBM team heavily upgraded the machine and in their second game Deep Blue managed to beat Kasparov $3\frac{1}{2}$-$2\frac{1}{2}$ [4].

Deep Blue uses and evaluation function which measures the '*goodness*' [19] of a given position based on four basic values; **material** which evaluates the worth of particular pieces, **position** looks at relative positions on the board, **King safety** gauges how much threat the King is under, and **tempo** is similar to position but more focused on developing greater control of the board.

AlphaGo is a machine developed by Google in London to play the ancient Chinese board game *Go* and became the first computer to beat a professional player without handicaps on a full sized 19x19 board in 2015. In 2016, AlphaGo went on to defeat Lee Sedol, a 9-dan professional champion, in a five-game series. Most notably, in 2017, AlphaGo was able win against Ke Jie, ranked number one globally at the time, in a three-game match.

Unlike Deep Blue, AlphaGo uses a *Monte Carlo* tree search with knowledge gained through machine learning techniques as opposed to evaluating all moves. *Go* is notoriously difficult to develop due to the sheer number of possible moves a turn which would require an unimaginable amount of processing power to compute. Instead, it uses the Monte Carlo tree search

[4] Half points are awarded to tied games

which is a heuristic to process decisions. It weights all the possible states in the game and only expands upon the most promising moves. Using thousands of past game-plays, the Google team trained two neural networks, a '*value network*' and a '*policy network*', which guided heuristic.

Since the neural networks were implemented using deep learning, they started off not knowing anything of the game of *Go*, not the rules or even how to move. As such, the computer was fully self-taught in the game without any human interference.

Chapter 2

Software Design

This chapter will examine the project in more depth, starting my discussing the project management aspect. Continuing, the overall design of the solution will be considered as to what the project needs to include and the tools that can be used. Finally, the design will be expanded to the actual implementation of the project talking about the software chosen and algorithms used.

2.1 Description of the Project

The project is a digital recreation of the age-old board game *The Royal Game of Ur* that is both easy to use and accessible for anyone to play. The game should also offer a computer opponent that has learnt to play through machine learning algorithms. The main task of this project is to **create an artificial intelligence opponent which can perform a well-informed moved based on its position on the board**.

2.1.1 Non-Functional Specification

- **Accessibility**: The software should be accessible for most users through ease of access and little-to-no set-up required

- **Documentation**: All the software should be comprehensively annotated to allow others to read and understand the code base

- **Extensibility**: The project should be able to expand to fit new requirements if desired

- **Flexibility**: The software should have many options to change how the game feels and plays
- **Maintainability**: The software should have maintenance if any issues are found so they can be fixed in a timely manner
- **Stability**: The software should run smoothly on all machines
- **Universality**: The product should be able to work on most major platforms, including Microsoft Windows, Macintosh OS X, and Linux Ubuntu
- **Usability**: The software should not have any major or game-breaking bugs and should run efficiently

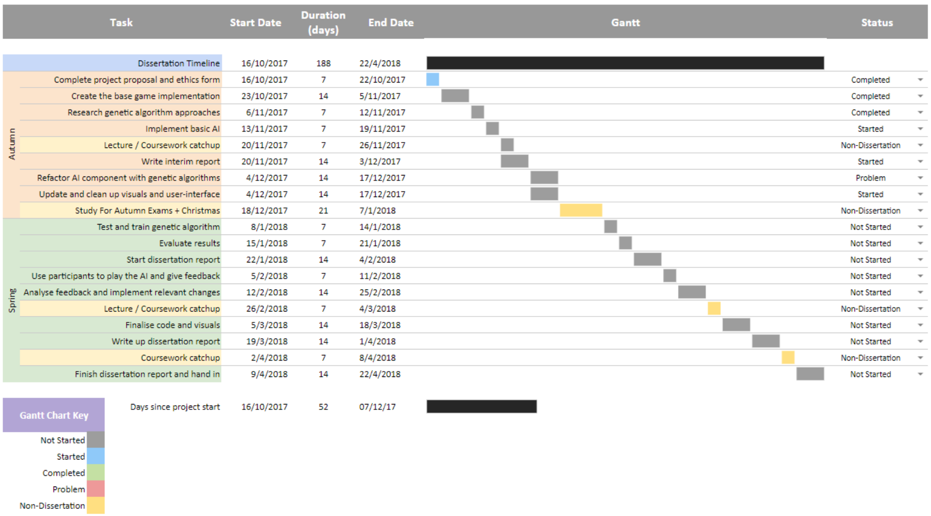
2.2 Methodology

This section will describe the techniques and tools that the project will use and other project management methods.

The project will follow an agile development model which uses rapid prototyping and continuous refinement of existing systems. The concept of this model is that each *sprint cycle* has a list of 'To-Do' tasks, or *user stories*, which should be completed by the end of the sprint. The project used a two-week cycle for the large tasks such as designing the game engine which were further broken down to one-week cycles to complete specific elements such as creating the game loop.

An example of the continuous refinement includes when designing the state machine where a basic implementation was first built in the initial cycle which has some rudimentary functionality but still accomplished the task of routing methods to an active state. In later cycles, more features were added such as the ability to pass mouse and key events through when they were needed. The agile model worked well for this project as it allowed for faster progress and only including features that were relevant rather than bloating the application.

The project will be using a Gantt chart to schedule tasks that the solution requires. This includes completing administration work, any research that is needed for the project, and the various stages to build the final product. The chart outlines building the game, researching then implementing the AI by the end of the Autumn term; then, in the Spring term, testing and evolving the AI and collecting any testing data before writing the dissertation report.



	Task	Start Date	Duration (days)	End Date	Gantt	Status
	Dissertation Timeline	16/10/2017	188	22/4/2018		
Autumn	Complete project proposal and ethics form	16/10/2017	7	22/10/2017		Completed
	Create the base game implementation	23/10/2017	14	5/11/2017		Completed
	Research genetic algorithm approaches	6/11/2017	7	12/11/2017		Completed
	Implement basic AI	13/11/2017	7	19/11/2017		Started
	Lecture / Coursework catchup	20/11/2017	7	26/11/2017		Non-Dissertation
	Write interim report	20/11/2017	14	3/12/2017		Started
	Refactor AI component with genetic algorithms	4/12/2017	14	17/12/2017		Problem
	Update and clean up visuals and user-interface	4/12/2017	14	17/12/2017		Started
	Study For Autumn Exams + Christmas	18/12/2017	21	7/1/2018		Non-Dissertation
Spring	Test and train genetic algorithm	8/1/2018	7	14/1/2018		Not Started
	Evaluate results	15/1/2018	7	21/1/2018		Not Started
	Start dissertation report	22/1/2018	14	4/2/2018		Not Started
	Use participants to play the AI and give feedback	5/2/2018	7	11/2/2018		Not Started
	Analyse feedback and implement relevant changes	12/2/2018	14	25/2/2018		Not Started
	Lecture / Coursework catchup	26/2/2018	7	4/3/2018		Non-Dissertation
	Finalise code and visuals	5/3/2018	14	18/3/2018		Not Started
	Write up dissertation report	19/3/2018	14	1/4/2018		Not Started
	Coursework catchup	2/4/2018	7	8/4/2018		Non-Dissertation
	Finish dissertation report and hand in	9/4/2018	14	22/4/2018		Not Started
	Days since project start	16/10/2017	52	07/12/17		

Gantt Chart Key	
Not Started	⬛
Started	🟦
Completed	🟩
Problem	🟥
Non-Dissertation	🟨

Figure 2.1: Gantt chart designed at the start of the project

A dependency chart had also been created which provides a list of tasks which need to be completed before other tasks can be started. This specific chart describes the major undertakings that are required for the implementation of the software. The chart was created to help with visualising the tasks as well as helping to inform the previously mentioned Gantt chart time-line. The chart also depicts aspects of the project which can be built simultaneously or parts than can be built any time after another task has been completed.

The testing for the final solution will be accomplished by asking volunteers to play against the various AIs and see which version is the most challenging. Each participant will then be asked to fill out a brief survey asking for their thoughts on the different opponents they face and if they noticed any differences in their play-style.

To keep the tests fair, they should be conducted blind so that each participant should not know which type of AI they are playing against. This will make the test more unbiased as the participants will have to judge each AI on their moves rather than applying pre-determined expectations on to the AI's moves.

The project will utilise a form of version control for the code-base. Version control is useful as it keeps track of all the changes throughout development and stores them off-site, therefore acting as an additional backup. Additionally, version control allows for code to be branched for updated ver-

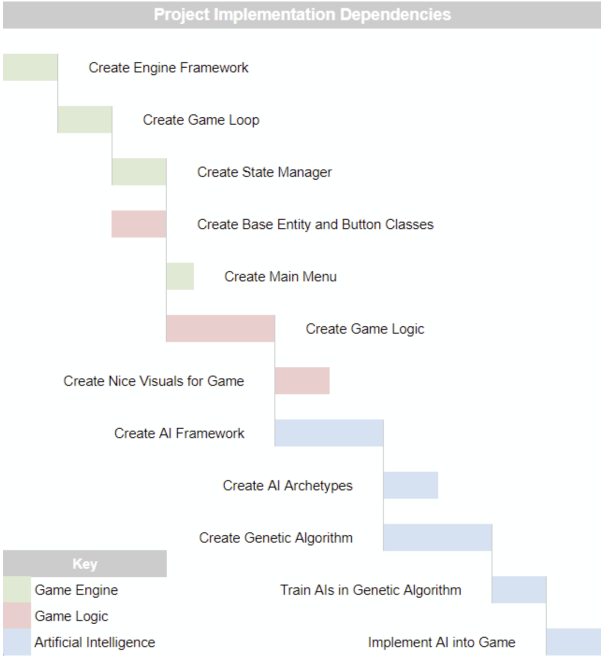


Figure 2.2: Dependency chart showing the order tasks need to be completed in

sions which means experimental features which may break the project can be kept separate until they are ready to be rolled back to the main branch.

Version control is also extremely helpful when debugging as the changes between each version are also stored so if a problem arises between two states, the differences can be analysed to help troubleshoot the issue.

2.3 Design

The following section will consider the assorted designs the project will need and different solutions for each of them.

Firstly, since the project is the development of a computer game, it should be able be playable. To accomplish this, the game would need to be able to update components and logic of the game, draw to the screen, and accept input from the user(s) as a minimum. These functions are all controlled in a *game loop* which is the backbone to almost all modern games [20]. A game loop is a block of code that runs continuously throughout the course of the game. Each loop (or cycle) of the game processes the user input without blocking the thread, updates the game's state, and finally renders the game [20].

Games often have many various aspects, each with their own distinct functionality; for example, a menu screen or diverse levels. To keep track of all the different pages, games often use a state system where each individual level would become a state that is solely responsible for its data and logic [21]. To control all these states and switching between them, a *state manager* would be needed.

A manager would oversee initialising, swapping, running, and destroying states. Another advantage of a manager is, instead of holding references to each state and hard-coding which state should be currently active, through polymorphic techniques, all calls to the active states can be sent through the manager. This allows for a far cleaner and efficient solution as if a state needs to be changed, simply a reference in the manager needs to be changed rather than using a giant switch method (see *figure 2.3*).

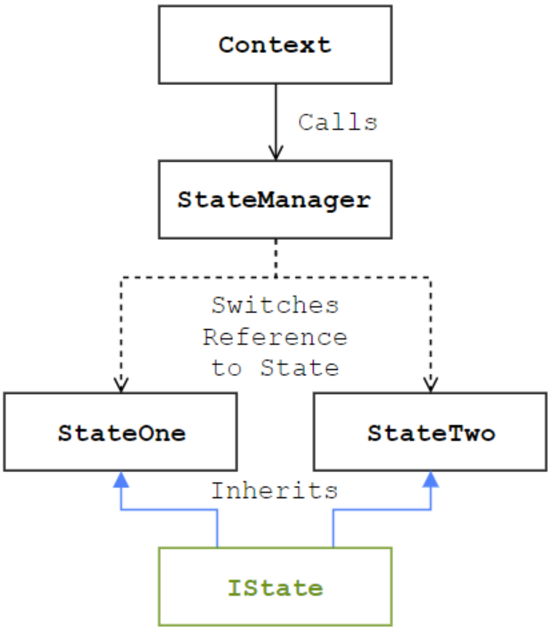


Figure 2.3: Simplified diagram of a state pattern

As such, the project should be coded using an *Object Oriented Programming* methadology. This allows for cleaner, more maintainable code base and quicker development over functional or procedural models. As a result, the complexity of the project can increase while still maintain - if not lowering - development time [22].

Object oriented programming has many benefits including *abstraction* which allows for obscuring the more complex logic that the current user (class/function) does not need to know; re-usability of object so, if three button are needed, the code does not need to be rewritten three time; and hierarchy which allows for object to have child object which inherit their methods and members but can overwrite or extend their functionality for more specific tasks [22] [23].

The project requires a graphical user interface over a simple text-based interface. Although creating the program in a terminal application is possible, the result would be a lot higher if the game was drawn on a *canvas*. A canvas is a bitmap graphical interface which can be drawn to through commands. Primitive shapes are usually built-in, such as rectangles and ellipses, but combined with the object orientated model, objects can create more complex shapes [24]. Canvas also allows for pixel perfect detection and manipulation, as well as an overall low level of control.

2.3.1 Artificial Intelligence

For the artificial intelligence, there are a few options for creating an AI that can play the game. One method is using minimaxing trees which calculates all possible states that the game could be in [15]. By organising the states in a graph structure, linking them by possible next moves, the graph can be traversed to analyse all potential future moves to choose the move with the best outcomes. As discussed before, other than very simplistic games, the number of permutations would require sizeable memory and high processing power. Aspects of this approach are usable though by, instead of pre-computing all states such as *Deep Blue*, analysing the current board and simulating future states to inform the final move while the game is running. With some efficiency methods like *AB Pruning* this method seems feasible but slow [15]. Additionally, since *The Royal Game of Ur* can have multiple paths into the same state, sometimes through circular routes, extra care would need to be made to prevent infinite looping.

Neural networks are also another viable solution as used in *AlphaGo*. By training the AI on past play-through data, the algorithms can develop logical neurons which try to match a move to previously analysed games. These types of systems unfortunately require more time and resources than have been allocated to this project and thus must be rejected for this paper.

A final approach would be to use an evolutionary algorithm which uses a population of AIs to play the game, each encoded with a unique DNA structure which informs how they play. Once all cases have been calculated, they are assessed and evaluated using a fitness function. This function ranks how well the AI was at competing the task. A new population is then chosen where each member is from two of the previous generation which are combined with the best traits of both, and the better their fitness score the higher the chance of being picked. Combined with the minimaxing algorithm, this design would gradually increase in intelligence over generations. Furthermore, should the need or requirements for more resources become available either during or after the project, the games that the genetic algorithms simulate can be used as past data for a neural network or feed directly into it as it learns.

By manually altering the DNA values, certain archetypes can be explored to see their effect on players. Examples of architecture could be an aggressive agent which always takes the opportunity to capture an opponent's counter, or a furthest first agent which just tries to get all its counter to the end, one by one.

Genetic Algorithms

A genetic (or evolutionary) algorithm is a meta-heuristic inspired by the process of biological natural selection and is a way to optimise solutions over the several iterations [25].

The basic process starts by **initialising** a population of workers which will perform the set task. There can be any number of workers, from dozens to thousands, all of which have their own DNA which is usually randomly generated at the start.

Once the task is performed, the next step is the **evaluation**. All workers are then evaluated as to how successful they were at completing their task which is calculated in a *fitness function.* The final part to this stage is to normalise all the workers' fitness values to one scale, so the best performer will have the maximum score and the worst the lowest. This allows the fitter workers to have a higher chance of being chosen for the next step while the workers who did not do well to be discarded.

The **crossover** is where two of the previous workers are chosen (weighted towards the better candidates) and their DNA is combined. This creates an offspring which will join the population of the next workers while, theoretically, inheriting the best traits from each of its parents.

The last step is arguably the most crucial. Before the offspring workers begin, a **mutation** occurs to their DNA. At a low percentage change (typically 0.1% but depends on the task) some of the DNA will be replaced by random generation. This step allows for more variety in the process and allows the workers to potentially escape local maxima[1], as well as the

[1] A local maxima the maximum value in a small area but may not be the true maximum of a function

algorithm not being dependent on the initial random generation.

Once all the steps are computed, they are repeated until either a predefined number of generations has been completed or there seems to be no improvement in the maximum fitness that is being obtained [10].

The advantages of genetic algorithms are they are brilliant at finding the most optimum route and optimising solutions. They are also faster than searching a large search space and require minimal memory requirements. Downsides, on the other hand, include that it can depend heavily on chance if there are many variables to consider [11] since the DNAs all start with randomised attributes. With a large enough population this can be minimised. Lastly, as mentioned, the heuristic can get stuck on local maxima but, again, with a large enough population and the mutation mechanic the algorithm should evolve out of the local peak after a few generation.

2.4 Implementation

Lastly, this section will cover the implementation of the design, as well as the tools used in building the project.

There are a few of design choices to fulfil the requirement of creating a game; the first is using a commertially avaliable game engine such as *Unity* [26], *Unreal Engine* [27] or *GameMaker Studio* [28]. All three engines mentioned are incredibly powerful tools for creating games with GameMaker Studio mainly focused on 2D game creation, Unreal Engine's fantastic 3D and animation software, and Unity with a mix of both 2D and 3D capabilities. Through some early prototypes of the project, these engines became bloated very quickly with all the libraries and assets which were required. Since the projects is supposed to be accessible these drawbacks became increasingly more prevalent.

Another solution is to create a custom engine from scratch. Although this would be more development costly, the complete control over bespoke software allows for stricter control over the system and greater flexibility in design, for instance implementing a custom AI would become more straightforward.

The next question to answer for building a bespoke game engine is to choose a programming language to code in. The two biggest languages that would be suitable for a project of this size and complexity are *Java* [29] and

C++ [30].

C++ offers many advantages such as its lightning fast speed and greater control over the machine it is running on. Java, on the other hand, has the advantage of being interpreted over compiled meaning it can be run on almost any machine, whereas C++ must be compiled to every specific operating system to run [31]. Since the project calls for an accessible solution, Java will be the programming language of choice.

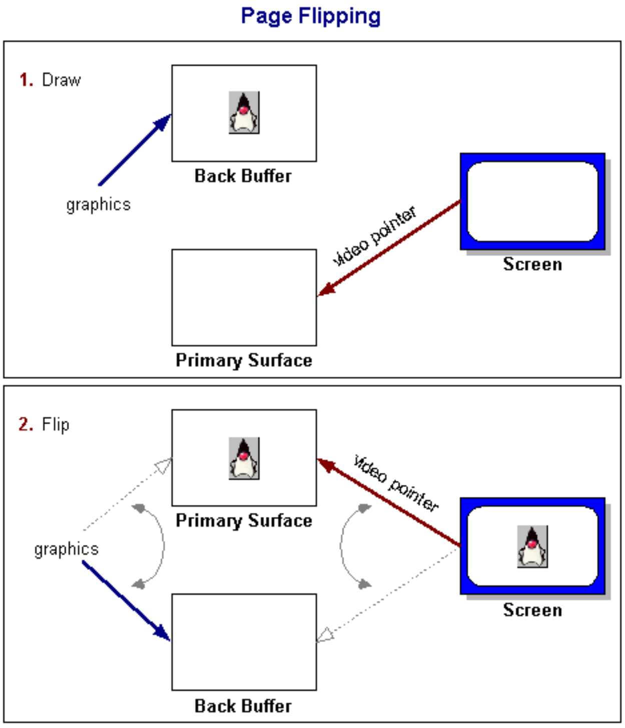


Figure 2.4: Double buffering example [29]

After the language was chosen, a Java GUI framework needed to be decided between *JavaFX* and *Swing*. JavaFX (or FXML) is brilliant at creating complex GUI elements easily by using an editor with styling tools and an array of pre-built components such as panes, scroll bars, buttons, labels and much more. Although these features are possible in Swing, it is far more difficult to implement and edit in the future should there be a need to alter the layout.

Nonetheless, Swing was chosen for the project because this application will use a canvas and it allows for greater control of its individual features through inheritance. In testing, the JavaFX canvas container did not interact well with commands issued from other threads whereas Swing has an asynchronous rendering method which creates a far more stable program.

The canvas will also be using a technique called *double buffering.* Double buffering is when the system allocates twice as much resources by creating two canvas elements. The first is used as normal and shown to the screen, but the second is the canvas that the system will draw to. Once completed drawing, the system will then switch which canvas is being drawn and then start drawing to the first canvas while the second is being shown on screen.

Now that the context is created, a game loop is needed to create a stable game which is responsible for getting user input, updating the game logic, and rendering the game to the canvas [20]. Most professional game engines separate the game's updating from its rendering to allow greater flexibility when running on different machines, such as, they tend to try to run the game logic at 30 time a second (*hertz*) and render the game to the screen 60 times a second (*Frames Per Second* or *FPS*).

An advantage of separating the game's updating from its rendering is that it allows the loop to be more flexible with different machines. For example, on a less capable computer that may not be able to render the screen as quickly as a machine with a powerful graphics card, the game loop is able to still update the game logic at the same rate but can vary the drawing calls per second. Although this may not run as smoothly to the eye, the game will still perform at the same speed so, in the case of multi-player games, the two machines can remain in-sync [20] [32]. As such the project's custom engine will run at 30 hertz and 60 FPS also.

The last aspect to consider is how to build the artificial intelligence system for the game. Since the project will be utilising a genetic algorithm, one of the first elements to decide is what each AI's DNA will contain. The DNA should contain each instance's specific coding that will improve the way it plays the game.

One way to accomplish this is by having the DNA contain weights for different scenarios that each counter could potentially encounter; for example, if a counter could land on a rosette which gives the player an extra turn, a rosette weight would be added to that counter's score. After all the weight for each counter have been calculated and normalised, the counter with the highest score should be the optimal move.

2.4.1 Game Engine Algorithms

The project contains numerous algorithms for all the various aspects of the solution. In this section, a few of the notable methods and structures will be discussed.

Game Engine

The state manager uses a state pattern to switch references to the current state. Most implementations of this design hold a pointer to the currently loaded state. In this project, the manager also holds an array of loaded states and a reference to the shown states. This allows for the system to keep more than a single state in memory so, for instance, if the game switched to a paused state, the contents of the main game would not be lost.

Another way to implement this approach would be to use a stack data structure where newer states get pushed and popped as needed. In testing, this approach appeared more restrictive as this formed a unidirectional model whereas the current solution allows for greater flexibility for switching between any state at will.

Since the states can be switched to in any order, a communication method was needed so that information could be passed from state to state. Therefore, a bundle system was created which allows for multiple data objects to be passed when initialising new states and switching between states.

The bundle system uses a hash map to keep track of the data being transferred. The map uses a string as the key which is used to identify the data; the data is stored in the value as a base object class. The key string used when sending data is the same string needed when getting the data in the new state.

The game loop works by constantly running a loop while the game has not been exited. In the loop, a sequential update method followed by a draw method sent through the state manager which, in turn, updates and draws the rest of the program. However, if the loop was as simple as described, the game would run incredibly inconsistently, and the product would feel poor.

Instead, a more complex loop was implemented which tracks the current time in nanoseconds and can call update and draw methods separately. The loop starts off by getting the current nanosecond from the system and store it in variables for the last time the game was both updated and rendered[2].

The loop then tries to update the game. The loop can update the game multiple times a per cycle which allows it to catch up on lost updates if the system had been caught up on another process for too long. This allows the game to keep a constant speed, regardless of other interferences.

After the game has updated, the game is then rendered to the screen. Before the render method is called, an *interpolation* value is calculated. Also referred to as *delta time*, this value is used to make the rendering appear to be smoother as it balances the time the CPU took to complete the frame and the time the game should be rendering at. For example, if the CPU

[2] The game loop also tries to use atomic objects where possible to make the loop that much more efficient

completed the frame quickly, then the delta time would be smaller. If the CPU took longer to render that frame, the rendering would be behind, so the delta time would be larger. This is useful as the rendering of objects can interpolate using the delta time and appear to be moving smoothly, rather than jerky, twitching behaviour.

Finally, the game loop yields processing time to the CPU if it can. This helps the system multi-task and keeps the game loop from monopolising the CPU.

Drawing to a Canvas

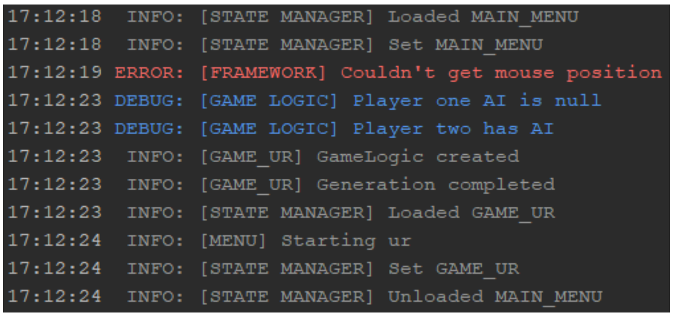
Since drawing can be quite complex for entities, all attributes were moved to an abstracted *base entity* class. This class keeps track of the object's dimensions, position and, crucially, the object's previous position. The previous position is used in conjunction with its current position, which is calculated after its update method has run, and the interpolation value passed in from the game loop. If the entity has moved, a new draw position is calculated which uses the distance between its last position and current position with the interpolation value determining how far it should move to appear smooth.

The base entity class is an abstract class which all other entities should extend from. Unlike other game engines where all entities are stored on the game loop layer, each state is responsible for creating, managing and deleting their own entities, GUI and widgets.

The canvas used is an array of pixels the width and height of the window created. Although each pixel can be individually coloured, most drawing operations use a combination of ellipses and rectangles which have built in methods in the built-in graphics interface. The canvas uses a Cartesian grid where the top-leftmost pixel is `(0, 0)` and the bottom-rightmost pixel is `(width-1, height-1)`. Using these coordinates, the positions of all on-screen elements can be represented using a *2-dimensional vector*.

Utility

Since vectors are heavily used when developing for canvases, a utility vector class was constructed to help abstract its complexities. This included methods that easily performed the addition and subtraction, as well as multiplication and division by a scalar value. In addition, more arduous mathematical functions are included; for instance, finding the magnitude of the vector, calculating the dot product between two vectors, returning the distance between two vectors, and returning a normalised version of the vector, along with many other functions.



```
17:12:18  INFO: [STATE MANAGER] Loaded MAIN_MENU
17:12:18  INFO: [STATE MANAGER] Set MAIN_MENU
17:12:19 ERROR: [FRAMEWORK] Couldn't get mouse position
17:12:23 DEBUG: [GAME LOGIC] Player one AI is null
17:12:23 DEBUG: [GAME LOGIC] Player two has AI
17:12:23  INFO: [GAME_UR] GameLogic created
17:12:23  INFO: [GAME_UR] Generation completed
17:12:23  INFO: [STATE MANAGER] Loaded GAME_UR
17:12:24  INFO: [MENU] Starting ur
17:12:24  INFO: [STATE MANAGER] Set GAME_UR
17:12:24  INFO: [STATE MANAGER] Unloaded MAIN_MENU
```

Figure 2.5: Snippet of application console with bespoke logger

A bespoke logger was also created for the project to allow for easier debugging and communication. The logger has several levels that can be called, from `trace` which is the lowest form of log to `error` which is the highest. These levels help identify the intensity of the logged message.

The logged messages are written to the application console with the time written, the importance level, a category and finally the message. With the addition that each level has its own custom colour, the output is clearly legible and informative. The text can be read easily, and the vital information is clearly visible, as seen in *figure 2.5*.

The console output can also restrict to a level and greater. For example, if the output level is set to warnings, only warning messages and above (errors) are printed. All messages are also written to a file on the user's system if needed for analysis.

2.4.2 Artificial Intelligence Algorithms

Genetic Algorithm

For the artificial intelligences, a DNA structure needed to be created to hold the weights for each of the different scenarios. This was completed by using a *chromosome* class which holds array of double floating-point numbers. The stored doubles refer to the weight of the scenarios that the chromosome represents.

Some chromosomes only have one value that they need to store, such as if the counter will land on a rosette, while other store multiple, such as the number of friendly counters already on the board. Therefore, the chromosome class was created to work effortlessly with either style by utilising method overloading. Method overloading allows for multiple definitions of the same method name to be created, with different arguments. This allowed for quick and easy creation of new chromosomes and automating loading the data in for the rest of the genetic algorithm[3].

The genetic process occurs in a simulation state, separate from the main game state. The algorithm used follows the same basic premise as most evolutionary algorithms - the first step is to **initialise** several agents. The implementation uses 100 instances, with random weights for each of the scenarios.

The next stage is to have the AIs play against each other and record notable statistics from the game, such as which instance won, how many turns the game lasted, and how many counters each player captured. These statistics are then used to **evaluate** that agent's fitness score which determines how successful it was. The current fitness calculation uses the score divided by the turn count, where its score is the amount of counter in the player's end area minus the number of counters in the opponents end area. This creates a range of values that heavily incentive winning in the quickest time, and with the fewest enemy counters completed to the end.

It was important, when designing the fitness function, that it remained as impartial as possible. This meant using as few statistics as possible, or statistics neutral to each player, to prevent manufactured tactics; for example, creating an aggressive AI due to capturing more enemy counters giving a higher fitness score.

Once all the fitness values have been calculated, a maximum fitness is computed, and all fitness values are normalised between 0 and 100. This is then used to populate a mating pool array-list where each AI is added to the list by the number of their normalised fitness. So, if an agent gets a normalised fitness of a 34, it is added to the mating pool 34 times. This method allows for each agent to have a chance to be picked for **crossover** based on their fitness score, where the most fit agent has the highest chance.

During crossover, the DNA of each chosen parents are combined to create a new child agent with a mixture their parents. There are multiple ways to calculate a cross-breed between two DNAs - the new value could be an average the parent values; a single point could be chosen where any value before that point one parents values are used, then values after use the others; or each value could be randomly decided case by case by flipping a coin.

The implementation created uses a multi-meme, hyper-heuristic which means that certain meta aspects of the genetic algorithm are encoded into the DNA itself. In this example, the crossover technique used is not hard-coded into the algorithm, instead each agent holds their own crossover technique. This technique is treated equally with the rest of the DNA values by

[3] To see a complete reference of all DNA scenarios and their descriptions, see *Appendix A*

evolving the best technique over the generations.

Finally, the new array of children agents goes through a **mutation** where each DNA value has a 1% chance of randomly changing to a new value. This helps keep the AIs from reaching a local maximum as it introduces new data that may not have been available in the randomly generated starting values.

To further this effort, 5% of all the agents in the list are completely randomly generated. This allows for new weights and challenges to be introduced to the genetic pool. These children AI then replace the original list of agents and a new generation is started.

The simulation is completed after a set number of generations has been run. The implementation ran until 1,000 generations had been completed, giving a plenty of time for the genetic algorithm to establish its results.

Evaluate Turn

Now that each AI has weights for all the programmed scenarios, they need to be used to evaluate a move. In the game logic class, when the next move is called, if the current player is an AI then they are asked to make a move from their current position.

After the dice has been rolled, all the agent's counters are analysed with the current roll to see which counters are able to move. Those counters are then put into a list and sent through to the AI to evaluate each available counter.

If the list is empty, meaning that there are no possible moves, the agent's turn is automatically ended and moved to the next player. Similarly, if there is only one possible move, that move is made, and the turn is ended. If there is more than one possible move, the algorithm will analyse which counter is the most optimal move using the weights from its DNA.

Firstly, an array of doubles is created with length of the number of counter. This will store the cumulative score that each counter receives. Then each counter is looped through, calculating its current position on the board, as well as its position on the board should it move.

The initial scenarios calculated are if the counters will enter any segments on the board, such as entering the board or the middle row. If so, the corresponding weights from the DNA are added to the score array at the counter's index.

Next, the algorithm goes through if the counter will land on a rosette tile, capture an enemy piece or exit the board. These are simple to calculate by querying the board with the tile that the counter will land on.

Then the scenarios of how many spaces ahead of an enemy counter, both before and after moving, are calculated. This was tricky to work out as the counter needed to know the opponent's route that the counter takes around the board. This is because if the enemy was still in their starting row, that would not appear in the player's route. Luckily, using some getters in the game logic class, the other player's route was obtainable.

Finally, the statistics for the number of each player's counters are calculated and added to each counter's score. Additionally, the furthest and closest counters get the respective scenarios weights added to them.

Once all the scores have been calculated, the counter with the highest score is the piece with the most optimal move given the current position of the board and the weight values encoded in the agent's DNA.

When prototyping the algorithm, it was found that the counter that hit the most scenarios was usually the counter which was chosen as the optimal move. In hindsight, this is logical since the more scenario hit the higher the score would be, regardless of how valued the scenarios are.

To combat this, another array is created, alongside the scores, which tracks how many scenarios each counter hits. At the end of the loop, before the best counter is chosen, all the counters have their scores divided by the number of counters they respectively hit. This normalised the values to allow the counter with a better move still compete with the counters with many smaller advantages.

AI Library

To store the various AI DNAs, a library class was created. This library is a simple static class which holds DNA objects with certain archetypes. For example, it holds the best agent after 1,000 generations for all three different evolutions, as well as the normalised average values for the final 100 agents in each evolution.

The class also holds some bespoke values which are used to simulate certain play-styles. For instance, there is an AI which is very aggressive and will prioritise capturing an enemy's counter over any other move. Another archetype is an agent which tried to get a counter to the end a quickly as possible before moving on to the next counter. This creates a very quick approach which is difficult to deal with as an opponent. On the other hand, another architecture tries to move all its counter up the board together. This may not be the most optimal approach, but it creates interesting style to play against.

The last archetype is a custom class which has weights designed to predict what the genetic algorithm may consider to be the optimal weights, based on human strategies. The weights were decided by observing how participants chose their moves and asking follow up questions to gain insight on why that move was prioritised over others.

Chapter 3

Appraisal

The contents of this chapter will evaluate the project and its outcomes, discussing the software's overall strengths and short-comings. The time and resource managements will be analysed, along with a self-refection on how the project progressed.

3.1 Evaluation

To train the AI agents, a genetic algorithm was used which is a school of machine learning. For this implementation, 100 agents were trained over 100 generations through an evolutionary process. Starting out with completely random attributes, each cycle two agents would play against each other. Their results would be recorded and used to calculate a fitness score which determines how well that specific agent was at completing the game. The fitness function used a formula which incentivised beating the opponent in as few turns as possible, as well as keeping the opponent from getting their counters to the end.

Once every agents' fitness was calculated, two agents were randomly chosen, weighted by their fitness value, and cross-bred to from a child agent which had traits from both parent AIs. This allows the population of agents to slowly grow and become better over the generations as the better agents have a higher chance to be chosen over the weaker agents.

Results of 100 Agents Over 100 Generations

The results of the agents are as follows (see *figure 3.1*), where each column represents a DNA value and the graph shows the spread of values, as well as the mean value and first to third quartile range[1].

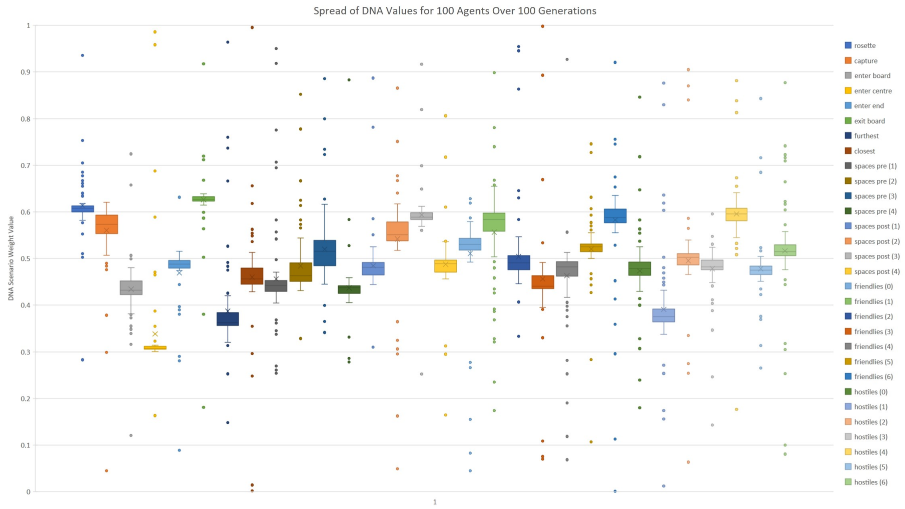


Figure 3.1: Distribution of DNA values of 100 agents after 100 generations

At first glance, the results show a massive spread of values ranging a complete spectrum of values. However, the quartile range of the scenario values are all narrow meaning that the genetic algorithm is working. Furthermore, there are still some interesting deductions that can be made from this data.

Firstly, the rosette scenario - which decides if it should land on a rosette tile - is very high and its quartiles are particularly narrow which indicates almost all the AI agree it is important. Similarly, the values for the scenario to exit the board has the highest mean value of all them and has a narrow quartile range implying that most of the agents have a similar value.

On the other side of the spectrum, the values for entering the centre row are, again, close together showing that the AI all agree on the value. This time the move is not being favoured and therefore has an exceedingly low value.

On the contrary, values such as `spaces pre (3)` which controls if a counter should move if it is three tiles ahead of an enemy counter *before moving*, have a rather broad range of values comparatively. This could be for a few reasons, namely that the scenario might not be that common which means that it wasn't a leading factor in winning or alternatively, that there were not enough generations to properly train all aspects of the agents.

[1] Points within the quartile range are not shown

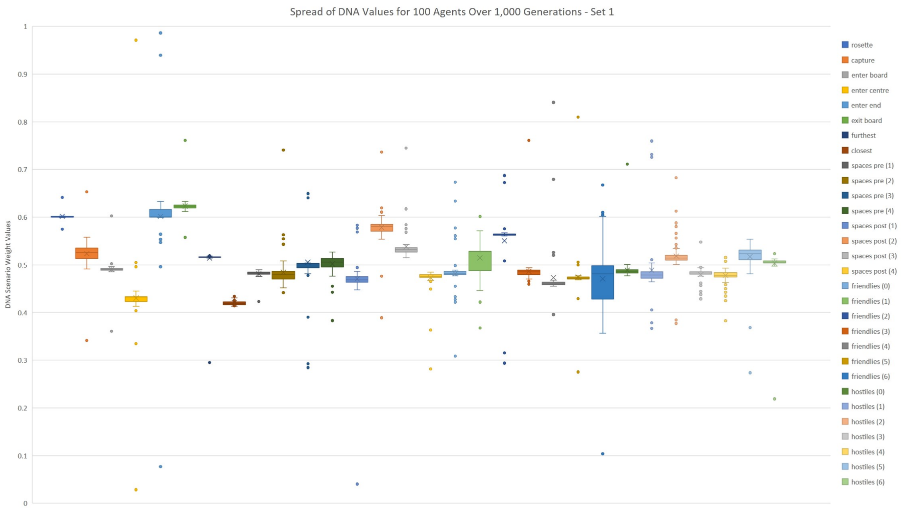


Figure 3.2: Distribution of DNA values of 100 agents after 1,000 generations

Results of 100 Agents Over 1,000 Generations

To remedy this, another test of the genetic algorithm was run, this time were calculated for 1,000 generations. The results for the set (see *figure 3.2*), have a noticeable improvement as almost all the scenarios have a must narrower range of values and there are far less outliers than the values in the 100 generation results. This mean that all the agents found a sustainable value for each scenario that is effective.

Interestingly, the graph shows many similarities to the 100 generation results, such as the rosette and exiting board values are still among the most prioritised, whereas entering the centre row is among the least. The former two can be attributed to common dominant strategies since landing on a rosette leads to an extra turn for the player and getting counters off the board are how the game is won. The agents not entering the centre row, on the other hand, could be contributed by counters then having the ability to be captured, resting progress, or it could be due to a strategy developed that has the AIs try to get as many counters on the starting section of the board first, before moving onwards.

The results also hold some surprises too. For instance, the scenario for having six friendly counters on the board has a very large range of values. This could be contributed to the scenario being unlikely and thus not refined as some of the other scenarios. Another likely reason is that the scenario doesn't particularly affect if a game is won or lost.

Cleaning Up the Data and Graphs

Although the graph for 1,000 generations is clearer, it is still not especially easy to read since the values are all clustered around 0.5. For that reason, a new graph was constructed with the aim to see the differences in the values more clearly. To accomplish this, the average value for each scenario was calculated and then normalised from 0 to 1. This gives a much more legible graph which clearly shows the values for each of the scenarios.

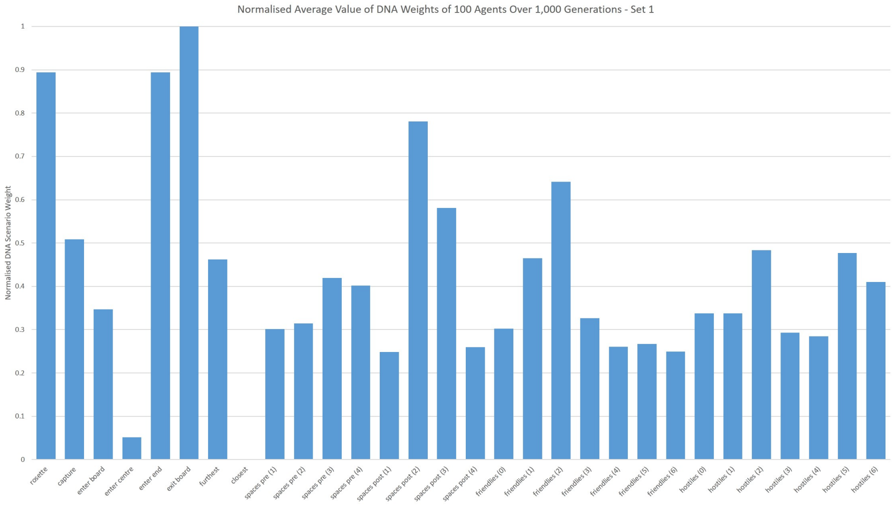


Figure 3.3: Normalised average values of DNA scenarios of 100 agents after 1,000 generations

By normalising the mean values, the weighing of each scenario can be clearly read and compared to other values. As evident, exiting the board is the aspect which the AIs prioritise the most while moving the counter closest to the start of the board is the least, perhaps due to trying to get counters to the end quicker rather than bringing all the counters up the board together in a pack.

On the other hand, some of the data still does not appear to make sense. For example, `spaces post (2)` represents a counter that will move in two tiles in front of an enemy counter and its mean value is substantially greater than the other weight values. This is intriguing because, due to the mathematical probability behind the dice used, two is the most common roll a player will get. This means that the AI is purposely putting their counter in danger as it is in the most likely space to be captured.

More Results of 100 Agents Over 1,000 Generations

Since genetic algorithms are highly dependent on random chance, the simulation for 100 agents of 1,000 generations was repeated another two times. This should help identify if some of the results from the first set were anomalies or specifics strategies developed through evolution.

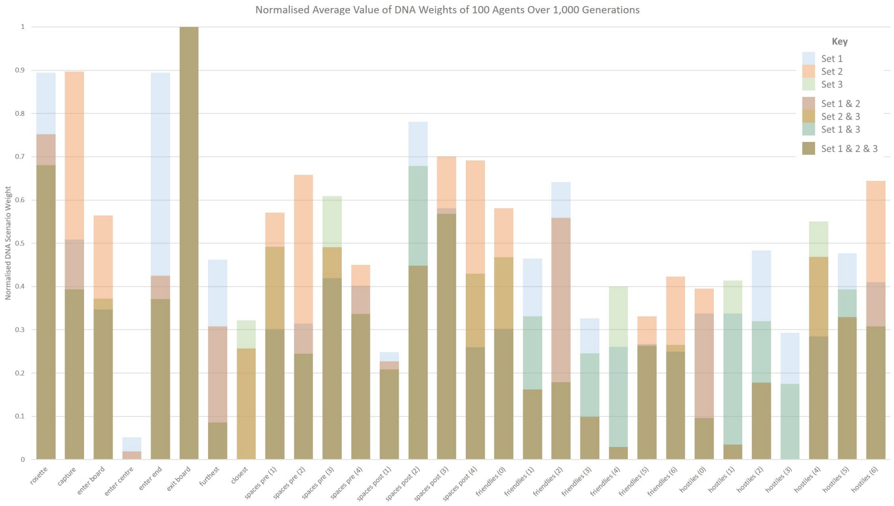


Figure 3.4: Normalised average values of DNA scenarios of 100 agents after 1,000 generations (overlaid)

The graph in *figure 3.4* shows the combined normalised average values of each of the three separate evolutions after 1,000 generations[2]. For the most part, the values for each of the sets seem similar. Exiting the board is the highest weighted scenario in all three evolutions which shows that it is essential and crucial for winning the game, which, from applied knowledge, is correct since all counters must have exited the board to complete the game.

Entering the centre is, again, the one of the lowest weighted values showing that entering the centre hinders performance and should only be done as a last resort. It is unclear why this strategy is so prevalent, but considering the results have been replicated, not once but twice, it proves that it is an effective approach.

Other scenarios, on the other hand, have a broad spread, `spaces post (4)` for example. This value is for the scenario a counter will land four tiles in front of an enemy counter after it has moves. This spread could be attributed to the event being quite rare in games. Additionally, early on a weighting could have been found worked which followed through to the end, either by it being a rare event or simply due to it not effecting the outcome

[2] To view all individual graphs, see *Appendix B*

as much - especially considering a four roll is the least likely outcome of the dice.

Ultimately, due to the nature of genetic and evolutionary algorithms, the reasoning behind why a particular weighting was found to be effective can only be speculated and taken into consideration.

Testing Architectures Against Humans

Now that some artificial intelligence agents have been created, they should be tested on human participants to see how they are in real situations.

Participants' skills ranged from novices who have never played the game before, to players who had strong knowledge of the game's rules. Each player would play against three separate architectures - an agent with weights designed to be aggressive, an agent with weights designed with bespoke values based on a human's intuition, and an agent with normalised average values from 100 agents evolved over 1,000 generations.

The agents were given in a random order and participants were not told which architecture they were playing against. This created an unbiased opinion from the players, as well as seeing the different strategies of the agents can be recognised.

The results of the experiments were interesting due to them being surprisingly consistent between all participants. All players found the aggressive agent to be the weakest of them all, yet the most frustrating. The players expressed annoyance that their counters were constantly getting captured, the AI held no other strategies for moves where there were no pieces to take. The moves of the agent were, for the most part, random and the participants were easily able to beat this architecture.

The next agent tested was the bespoke agent which was based on observing colleagues playing the game during the development process and analysing their insight on why they chose certain moves over others. Overall, participants found this agent to be far more challenging and intelligent than the previous aggressive agent. Player's games, on average, last the longest with this agent but the players ultimately won most of the time.

Finally, an evolved agent was competed against the participants. The results of the trails were like the bespoke agent, but the evolved agent ended up winning about half the games against the players. Participants almost unanimously agreed that this agent was the most fun to play as it felt almost as if human was playing against them. Since the players responded that the bespoke agent and the evolved agent felt similar, it seems the weights for the bespoke were successfully chosen, although not perfect.

3.2 Project Management

Agile development was used in the implementation stage of the project. Agile development is broken down into small sprints where a certain number of features are set to be added, or bugs to be fixed, or research to be conducted. This was helpful as it provided some organisation to the project, as well as giving a sense of accomplishment at the end of a sprint to see the results of the previous week and adding value to the product on a weekly basis.

The agile nature of the project also encouraged smaller, experimental prototyping of features to see their merit before being added to the main project. Version control played a big part in this as it allows the development of branches which are forks of the code base which can be modified with new features. Once they are implemented, the branch can either be merged back to the master branch and added to the project or discarded and the original master branch reverted back to before the fork.

This project used a Gantt chart to predict and track the schedule that each section of the solution would need to take to complete.

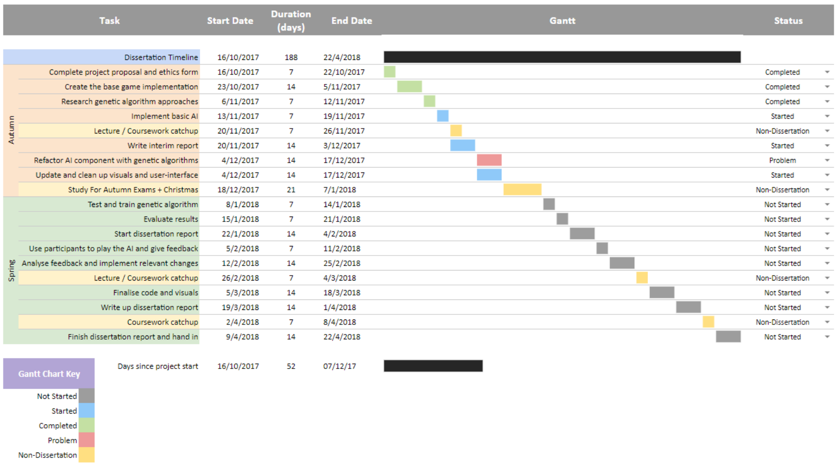
At the start of the project in October, a Gantt chart was made to illustrate a potential work schedule (*figure 3.5*). Throughout the development cycle, progress was stalled at several points due to lack of research and implementation taking longer than anticipated.

To overcome this, an updated Gantt chart (*figure 3.6*) was created at the start of the Spring term, early January. The new chart allowed more time for research and rapid prototyping during the spring term. The reasoning behind this is because substantial portions of the code-base did not need to be re-factored if the algorithm did not work or a better alternative solution was found. Additionally, more time had been allotted to each sprint as to not rush development which required more time in the end to fix all the problems which inevitably arise.

The revised Gantt chart worked successfully, and the project kept within its timelines. Having more time set aside for researching greatly increased the quality work within the project. The chart mostly estimated the time it would take for the challenges in each section to be completed accurately, with some minor exceptions where two tasks were assigned during the same time frame. These tasks were not completed at the same time but instead the time was divided between the task, completing them in serial rather than parallel. The amended design philosophy used after the second iteration of the timeline, created a more modular product. This allowed distinct aspects of the software to be altered without needing to change unrelated sections. Overall, I feel this created a more unified solution as different modules could be upgraded or changed at will and allowed for faster development.

Overall, I believe time and resources managements were used effectively. Before January, time was spent building the base game and creating a suitable backbone to support the main game and AI systems. The game logic was partially completed, but sections kept needing to be rewritten as the structure of the project became more complex.

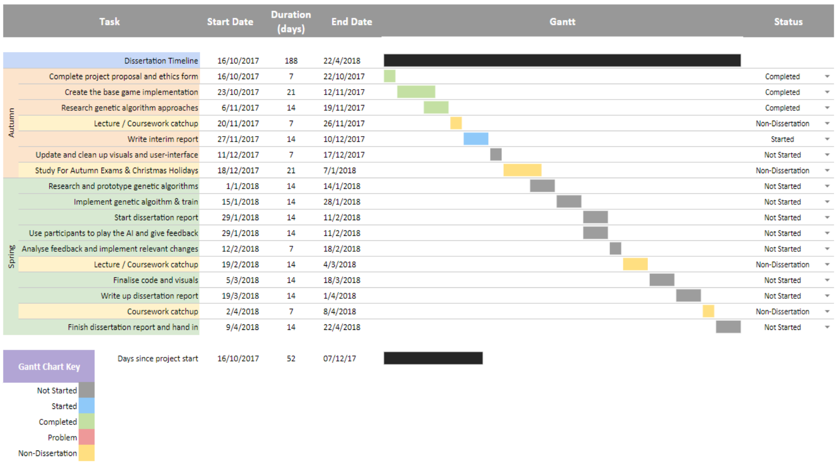
After January, and after I had analysed my progress so far, I realised that I kept rushing into adding new features without properly considering the entire project. I then laid out a new Gantt chart which helped focus my efforts and designed a more considered approach to developing the project. Instead of rushing into every situation, I found it more helpful, in the long run, to consider the whole project and plan each stage. This also helped in the short term to give each sprint cycle more structure and made each feature feel like a part of a larger solution, rather than an aimless objective.



	Task	Start Date	Duration (days)	End Date	Gantt	Status
	Dissertation Timeline	16/10/2017	188	22/4/2018		
Autumn	Complete project proposal and ethics form	16/10/2017	7	22/10/2017		Completed
	Create the base game implementation	23/10/2017	14	5/11/2017		Completed
	Research genetic algorithm approaches	6/11/2017	7	12/11/2017		Completed
	Implement basic AI	13/11/2017	7	19/11/2017		Started
	Lecture / Coursework catchup	20/11/2017	7	26/11/2017		Non-Dissertation
	Write interim report	20/11/2017	14	3/12/2017		Started
	Refactor AI component with genetic algorithms	4/12/2017	14	17/12/2017		Problem
	Update and clean up visuals and user-interface	4/12/2017	14	17/12/2017		Started
	Study For Autumn Exams + Christmas	18/12/2017	21	7/1/2018		Non-Dissertation
Spring	Test and train genetic algorithm	8/1/2018	7	14/1/2018		Not Started
	Evaluate results	15/1/2018	7	21/1/2018		Not Started
	Start dissertation report	22/1/2018	14	4/2/2018		Not Started
	Use participants to play the AI and give feedback	5/2/2018	7	11/2/2018		Not Started
	Analyse feedback and implement relevant changes	12/2/2018	14	25/2/2018		Not Started
	Lecture / Coursework catchup	26/2/2018	7	4/3/2018		Non-Dissertation
	Finalise code and visuals	5/3/2018	14	18/3/2018		Not Started
	Write up dissertation report	19/3/2018	14	1/4/2018		Not Started
	Coursework catchup	2/4/2018	7	8/4/2018		Non-Dissertation
	Finish dissertation report and hand in	9/4/2018	14	22/4/2018		Not Started
	Days since project start	16/10/2017	52	07/12/17		

Gantt Chart Key	
Not Started	▨
Started	🟦
Completed	🟩
Problem	🟥
Non-Dissertation	🟨

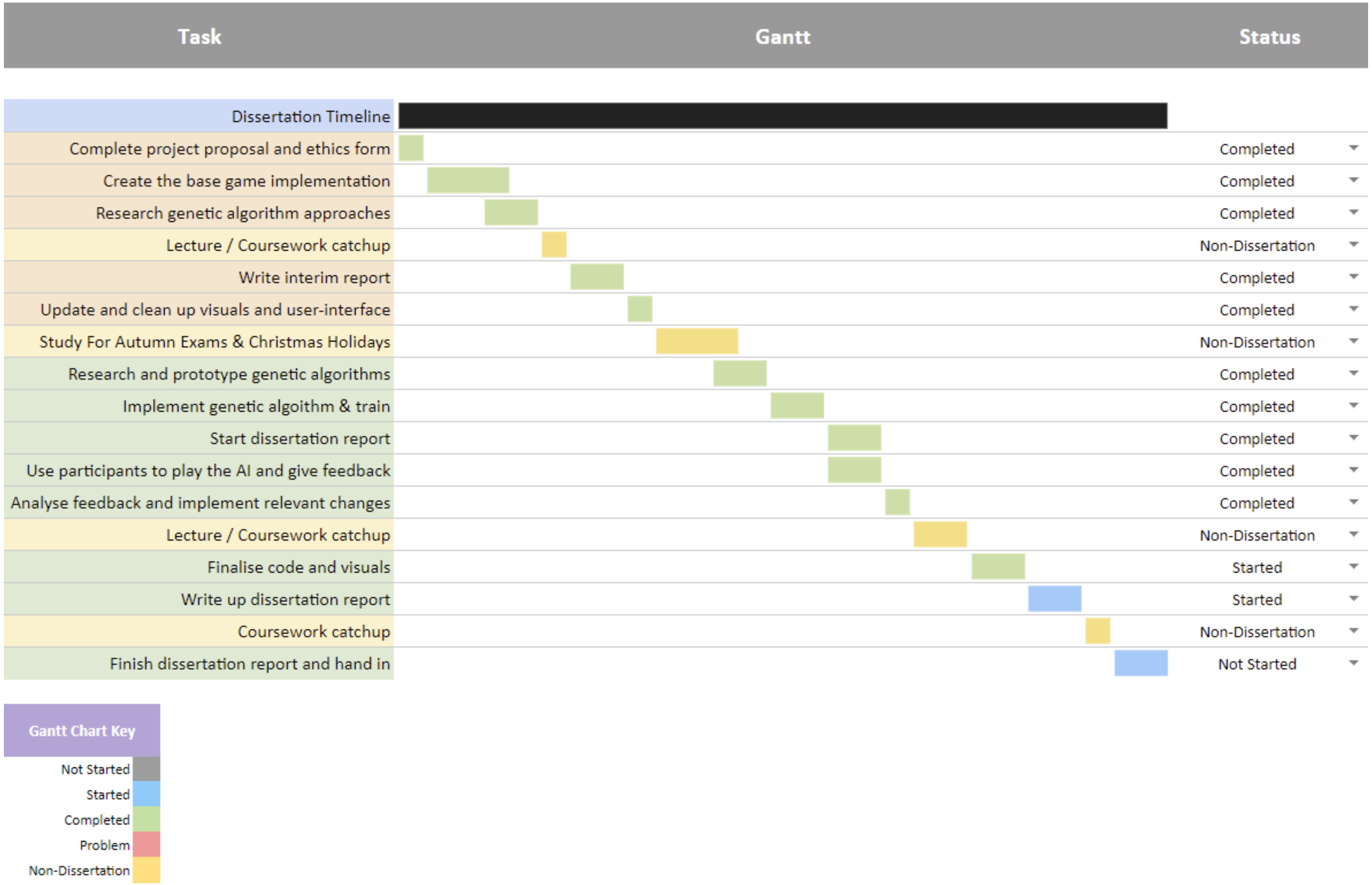
Figure 3.5: Gantt chart as of project proposal - half way through Spring term



	Task	Start Date	Duration (days)	End Date	Gantt	Status
	Dissertation Timeline	16/10/2017	188	22/4/2018		
Autumn	Complete project proposal and ethics form	16/10/2017	7	22/10/2017		Completed
	Create the base game implementation	23/10/2017	21	12/11/2017		Completed
	Research genetic algorithm approaches	6/11/2017	14	19/11/2017		Completed
	Lecture / Coursework catchup	20/11/2017	7	26/11/2017		Non-Dissertation
	Write interim report	27/11/2017	14	10/12/2017		Started
	Update and clean up visuals and user-interface	11/12/2017	7	17/12/2017		Not Started
	Study For Autumn Exams & Christmas Holidays	18/12/2017	21	7/1/2018		Non-Dissertation
Spring	Research and prototype genetic algorithms	1/1/2018	14	14/1/2018		Not Started
	Implement genetic algoithm & train	15/1/2018	14	28/1/2018		Not Started
	Start dissertation report	29/1/2018	14	11/2/2018		Not Started
	Use participants to play the AI and give feedback	29/1/2018	14	11/2/2018		Not Started
	Analyse feedback and implement relevant changes	12/2/2018	7	18/2/2018		Not Started
	Lecture / Coursework catchup	19/2/2018	14	4/3/2018		Non-Dissertation
	Finalise code and visuals	5/3/2018	14	18/3/2018		Not Started
	Write up dissertation report	19/3/2018	14	1/4/2018		Not Started
	Coursework catchup	2/4/2018	7	8/4/2018		Non-Dissertation
	Finish dissertation report and hand in	9/4/2018	14	22/4/2018		Not Started
	Days since project start	16/10/2017	52	07/12/17		

Gantt Chart Key	
Not Started	▨
Started	🟦
Completed	🟩
Problem	🟥
Non-Dissertation	🟨

Figure 3.6: Gantt chart for interim report - Autumn term and altered prediction for Spring term



Task	Gantt	Status
Dissertation Timeline		
Complete project proposal and ethics form		Completed
Create the base game implementation		Completed
Research genetic algorithm approaches		Completed
Lecture / Coursework catchup		Non-Dissertation
Write interim report		Completed
Update and clean up visuals and user-interface		Completed
Study For Autumn Exams & Christmas Holidays		Non-Dissertation
Research and prototype genetic algorithms		Completed
Implement genetic algoithm & train		Completed
Start dissertation report		Completed
Use participants to play the AI and give feedback		Completed
Analyse feedback and implement relevant changes		Completed
Lecture / Coursework catchup		Non-Dissertation
Finalise code and visuals		Started
Write up dissertation report		Started
Coursework catchup		Non-Dissertation
Finish dissertation report and hand in		Not Started

Figure 3.7: Final Gantt chart at the end of the dissertation

3.3 Reflections

In this section, I will go through all the aspects of the dissertation and discuss how they were successful and how I could improve on them if they could be redone.

The methodology went quite well, from my point of view. The use of an agile development cycle and the usage of Gantt charts really allowed me to quickly implement features into the software and remain organised. Although the project had a rocky start, once I realised the importance of these software engineering practices I became more comfortable working with a project of this size and complexity.

On the design side, I wish I had spent more time planning the steps to achieve the best product, rather than rushing into to development early. Before this project, I had not had any experience with a project this complex working by myself and would start writing code as early as possible since I knew the consequences of having to rewrite sections was trivial. It was only at the half way point of the project, when the AI system started to be developed, that I ran into issues refactoring sizeable portions of my code.

This was the motivation that forced me to reassess my design philosophy and become more organised in planning. Once I had gone back and planned out how features would work and fit together, the project when smoothly with no hiccups in research or development, other than from external aspects.

I found the implementation of the project to be straight forwards as my background is in software development. Although the structure of project had to be refactored several times, the individual algorithms and class created largely remained the same.

3.4 Further Development and Future Projects

If the project deadline was extended by a few weeks or a month, a stretch goal implemented to improve the product would be to redesign the way the AI agents play each other in the genetic algorithm. In the current build, two agents are paired up and play one game with one another and the their fitnesses are computed.

I believe a better way to design this problem would be to have each agent play every other agent and for multiple games. This would improve the rate at which agents evolve and would help create a more even array of agents. The down side to building the simulation this way, is the time for each generation would be greatly increased making the entire genetic process take exponentially longer.

Another aspect I would like to have improved is the overall game feel of the project. Much of the focus was made to make the AI aspect of the software to work well but I would have liked to add more quality-of-life features to the game. This would include adding menu options to change the players' names and colours, choosing which version of the AI to play, and adding sounds, music and improving visuals of the product. For example, one element which is lacking from the main game is a victory screen after the game has been completed.

If the project deadline was extended by a year, a new goal would be to modify the genetic algorithm to feed into a neural network. One of the short-comings a genetic algorithm is that each aspect of the game needs to be hard coded into the algorithm.

A neural network, on the other hand, does not need to know about any of the workings of the game and has a complete view of the board. If each tile and the starting and end areas of each player can be mapped to a simplified image, the entire board could be represented in a 8 by 7 pixel image. This image can be used as the input for the neural network where the pixel colour value represents if a counter is on that tile; for example, black could show no counter, grey could be player one, and white player two. The neural network could analyse the values of each pixel to inform its internal nodes. Additionally, each pixel only uses two bytes (black, dark grey, light grey, and white) making the input very efficient.

The genetic algorithm could then be replaced, and simulation would feed into the new network and use techniques such as *back propagation* to tell the network if the AI was successful which it can use to evolve its understanding of the game.

My biggest take-away from completing the project is how important proper planning is for a project. I've learnt to take my time in the preliminary stages of the project to outline the entire solution and work out how all the pieces fit together before rushing in.

Bibliography

[1] J. Botermans, *The Book of Games: Strategy, Tactics & History*. Sterling, 2008.

[2] T. B. Museum, "The british museum - the royal game of ur." http://www.britishmuseum.org/research/collection_online/collection_object_details.aspx?objectId=8817&partId=1, 2017. Accessed: 2017-OCT-23.

[3] Alphabet, "Alphabet." https://abc.xyz/, 2017. Accessed: 2017-DEC-01.

[4] Google, "Google play." https://play.google.com/. Accessed: 2018-APR-16.

[5] Google, "Google photos." https://photos.google.com/. Accessed: 2018-APR-16.

[6] Google, "Google maps." https://www.google.com/maps. Accessed: 2018-APR-16.

[7] Google, "Youtube." https://www.youtube.com/. Accessed: 2018-APR-16.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org, Accessed: 2017-DEC-02.

[9] S. Russell and P. Novig, *Artificial Intelligence - A Modern Approach (3rd ed.)*. Pearson Global Edition, 2016.

[10] L. Chambers, *The Practical Handbook of Genetic Algorithms Applications (2nd ed.)*. Chapman & Hall/CRC, 2001.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, 1989.

[12] BoardGameGeek, "Boardgamegeek." https://boardgamegeek.com/, 2000. Accessed: 2018-APR-18.

[13] A. M. Turing, *Computing Machinery and Intelligence.* 1950.

[14] S. Haykin, *Neural Networks and Learning Machines (3rd ed.).* Pearson Global Edition, 1993.

[15] I. Millington and J. Funge, *Artificial Intelligence for Games (2nd ed.).* Morgan Kaufmann, 2009.

[16] P. Talwalker, *The Joy of Game Theory: An Introduction to Strategic Thinking.* Presh Talkwalker, 2013.

[17] B. Shourd, "Thing i learned today: Number of tic-tac-toe boards." `http://brianshourd.com/posts/2012-11-06-tilt-number-of-tic-tac-toe-boards.html`, 2012. Accessed: 2017-DEC-03.

[18] F.-H. Hsu, *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion.* Princeton University Press, 2002.

[19] I. Reseach, "How deep blue works." `https://www.research.ibm.com/deepblue/meet/html/d.3.2.html`. Accessed: 2017-DEC-03.

[20] R. Nystorm, *Game Programming Patterns.* Genever Benning, 2014.

[21] A. Lewis, "Managing game states in c++." `http://gamedevgeek.com/tutorials/managing-game-states-in-c/`. Accessed: 2018-APR-16.

[22] A. O. LLC, "Lecture notes on object-oriented programming - motivation for oo." `https://atomicobject.com/resources/oo-programming/introduction-motivation-for-oo`. Accessed: 2018-APR-16.

[23] A. J. Alexander, *Introduction to Java and Object Oriented Programming for Web Applications.* devdaily.com, 2009.

[24] K. Mowery and H. Shacham, "Pixel perfect Fingerprinting canvas in html5," Master's thesis, Univerity of California, 2012.

[25] M. Mitchell, *An Introduction to Genetic Algorithms.* Bradford Book, 1996.

[26] Unity, "Unity - game engine." `https://unity3d.com/`. Accessed: 2017-DEC-04.

[27] E. Games, "Unreal engine." `https://www.unrealengine.com/`. Accessed: 2017-DEC-04.

[28] Y. Games, "Gamemaker studio." `https://www.yoyogames.com/gamemaker`. Accessed: 2017-DEC-04.

[29] Oracle, "Java." https://www.java.com/. Accessed: 2017-DEC-06.

[30] S. C. Foundation, "Standard c++." https://isocpp.org/. Accessed: 2018-APR-16.

[31] U. G. Inc., "Java vs c++: Which language is right for your software project?." https://www.upwork.com/hiring/development/java-vs-c-which-language-is-right-for-your-software-project/. Accessed: 2018-APR-16.

[32] E. Delventhal, "Game loops!." https://www.java.com/. Accessed: 2018-APR-16.

Appendices

A List of DNA Values

`rosette` If a counter would land on a rosette

`capture` The counter will capture an enemy counter

`enter board` The counter will enter the board from the starting pile

`enter centre` The counter will enter the middle row

`ender end` The counter will enter the end tiles

`exit board` The counter will exit the board to the ending pile

`furthest` The counter is the furthest into the route (closest to the end)

`closest` The counter is in the starting area or the closest on the board to it

`spaces pre (1..4)` The counter is that number of tiles ahead of an enemy counter *before* it has moved

`spaces post (1..4)` The counter is that number of tiles ahead of an enemy counter *after* it has moved

`friendlies (0..6)` That many friendly counters are currently on the board

`hostiles (0..6)` That many hostile (enemy) counters are currently on the board

B Graphs of Genetic Algorithm Results

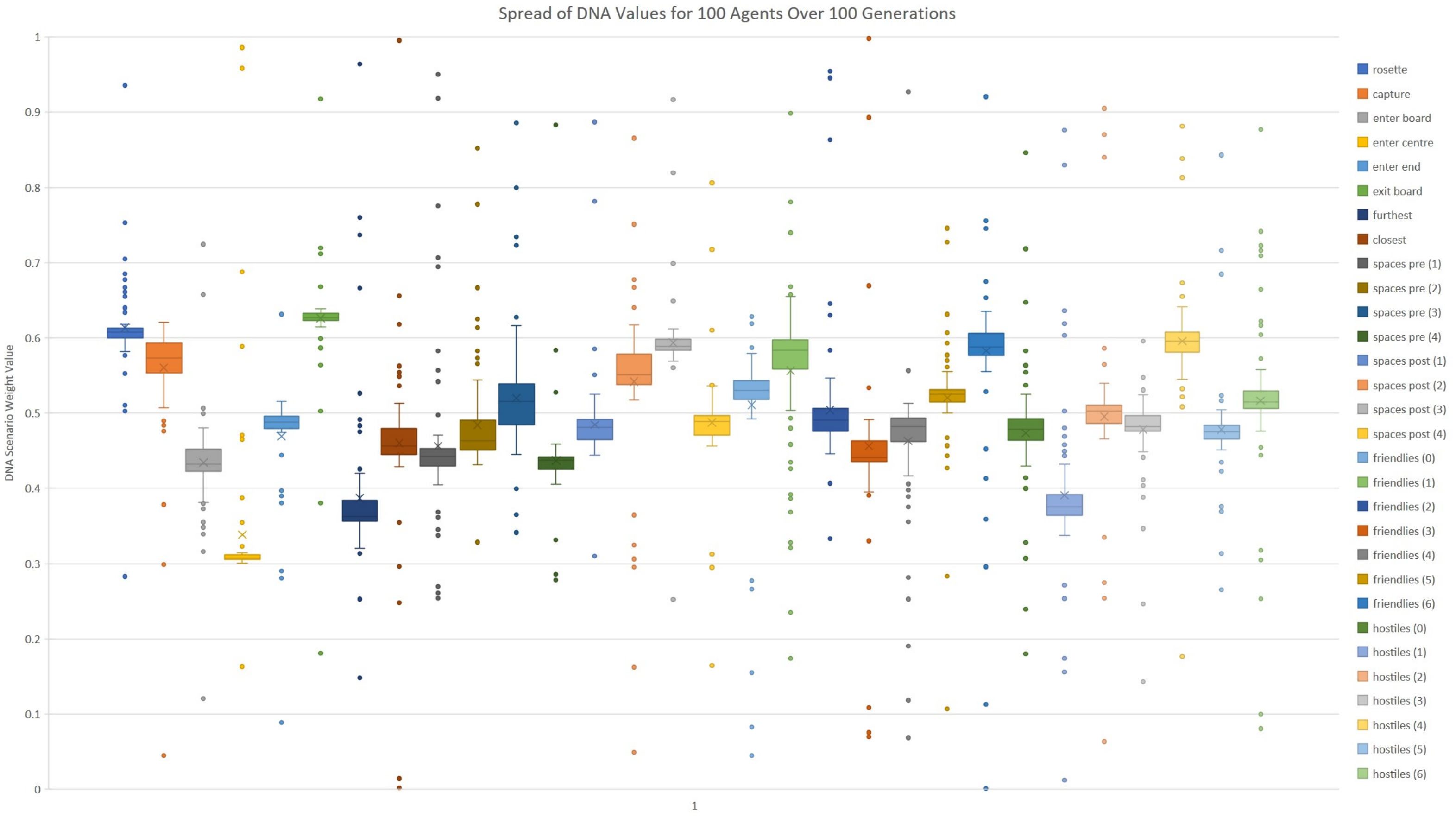


Figure B.1: Distribution of DNA values of 100 agents after 100 generations

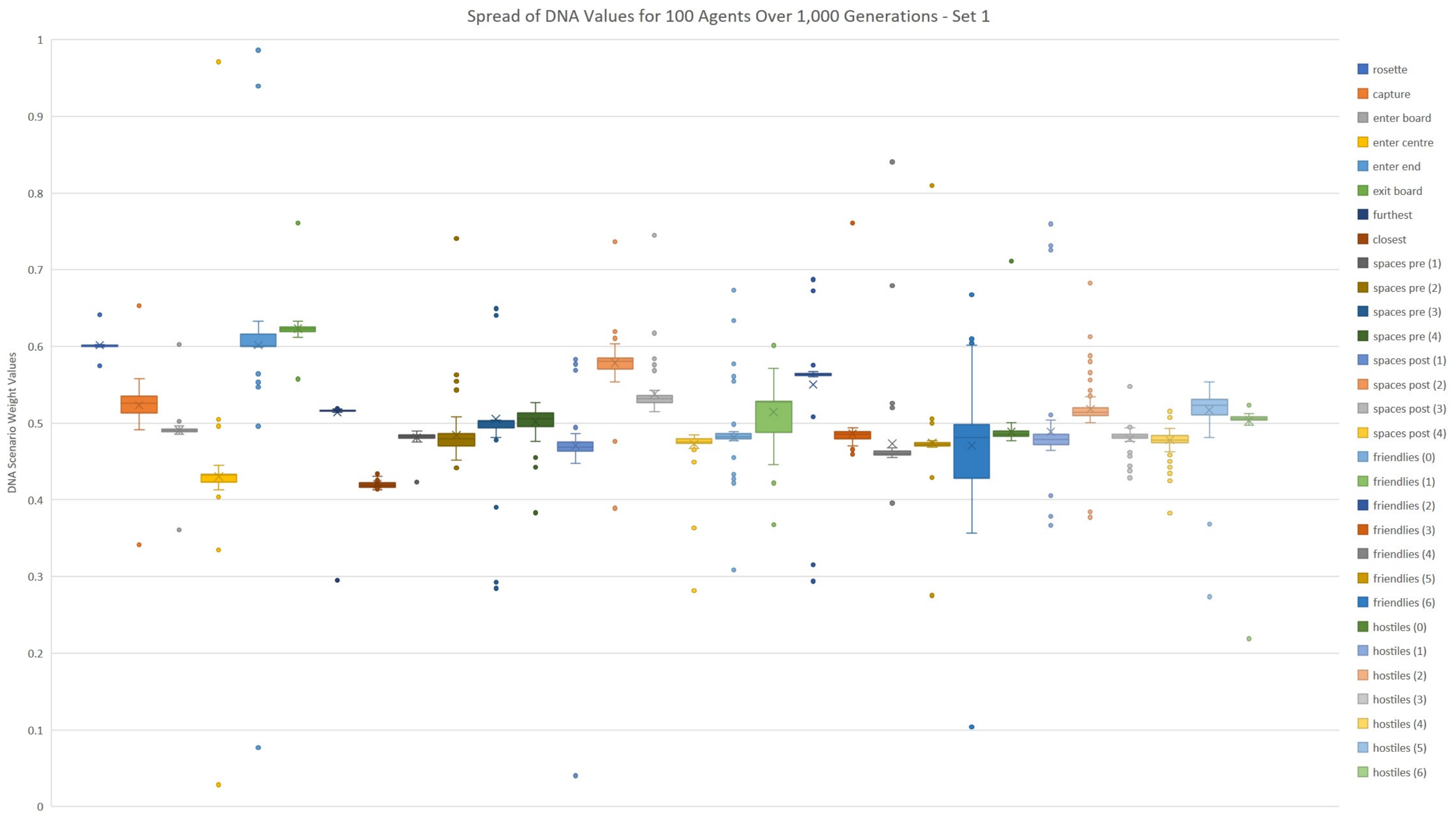


Figure B.2: Distribution of DNA values of 100 agents after 1,000 generations - Set 1

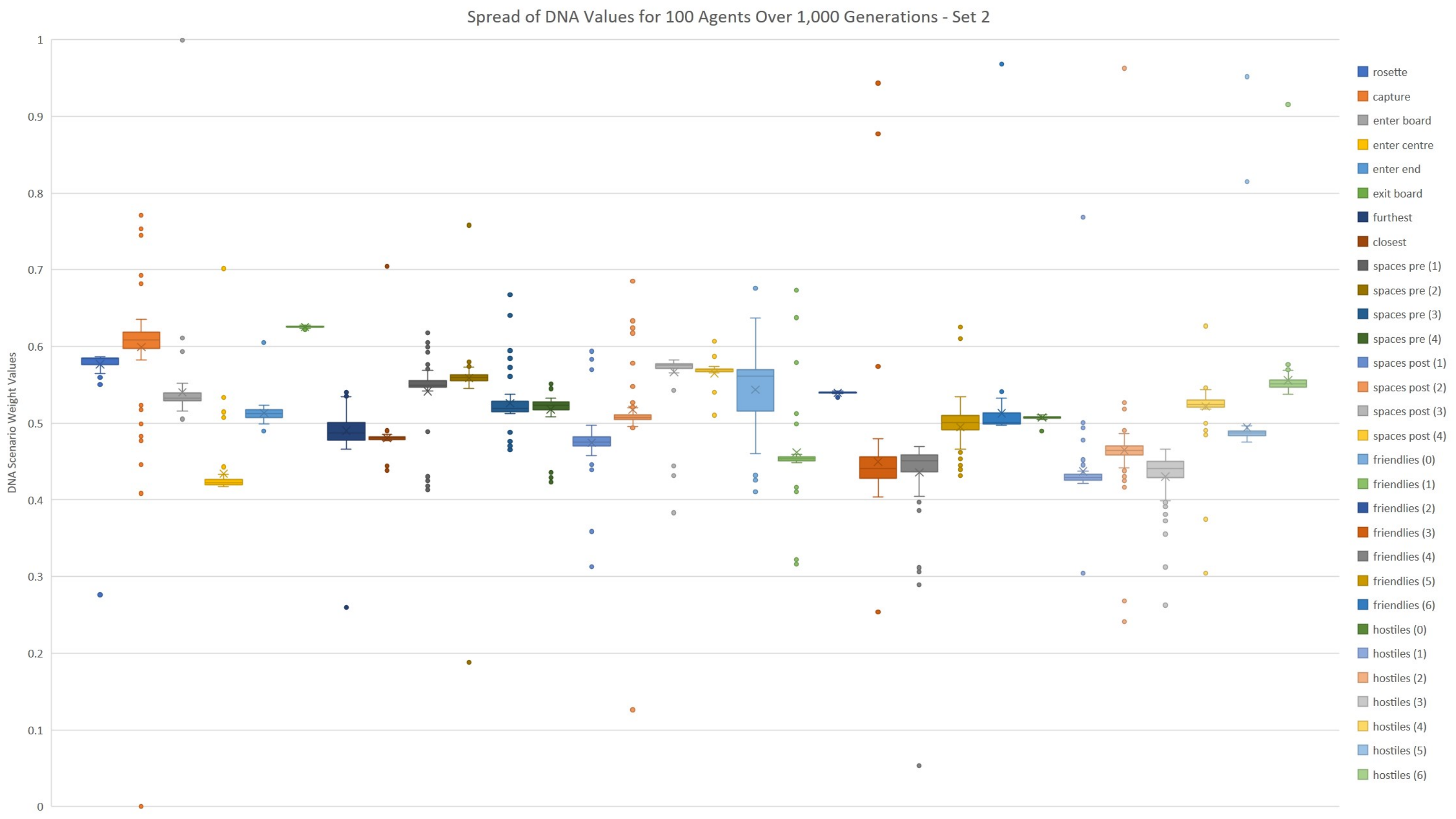


Figure B.3: Distribution of DNA values of 100 agents after 1,000 generations - Set 2

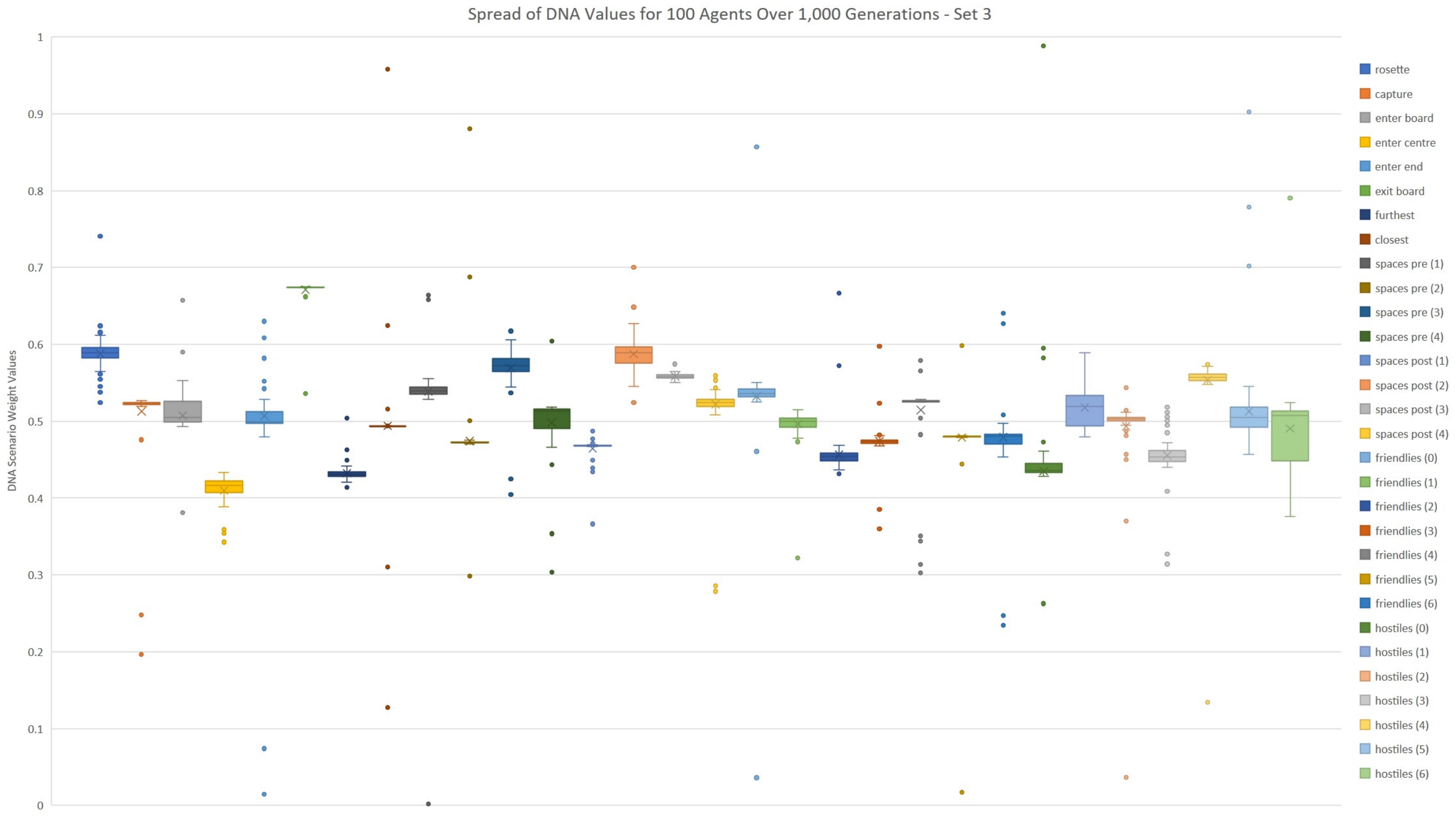


Figure B.4: Distribution of DNA values of 100 agents after 1,000 generations - Set 3

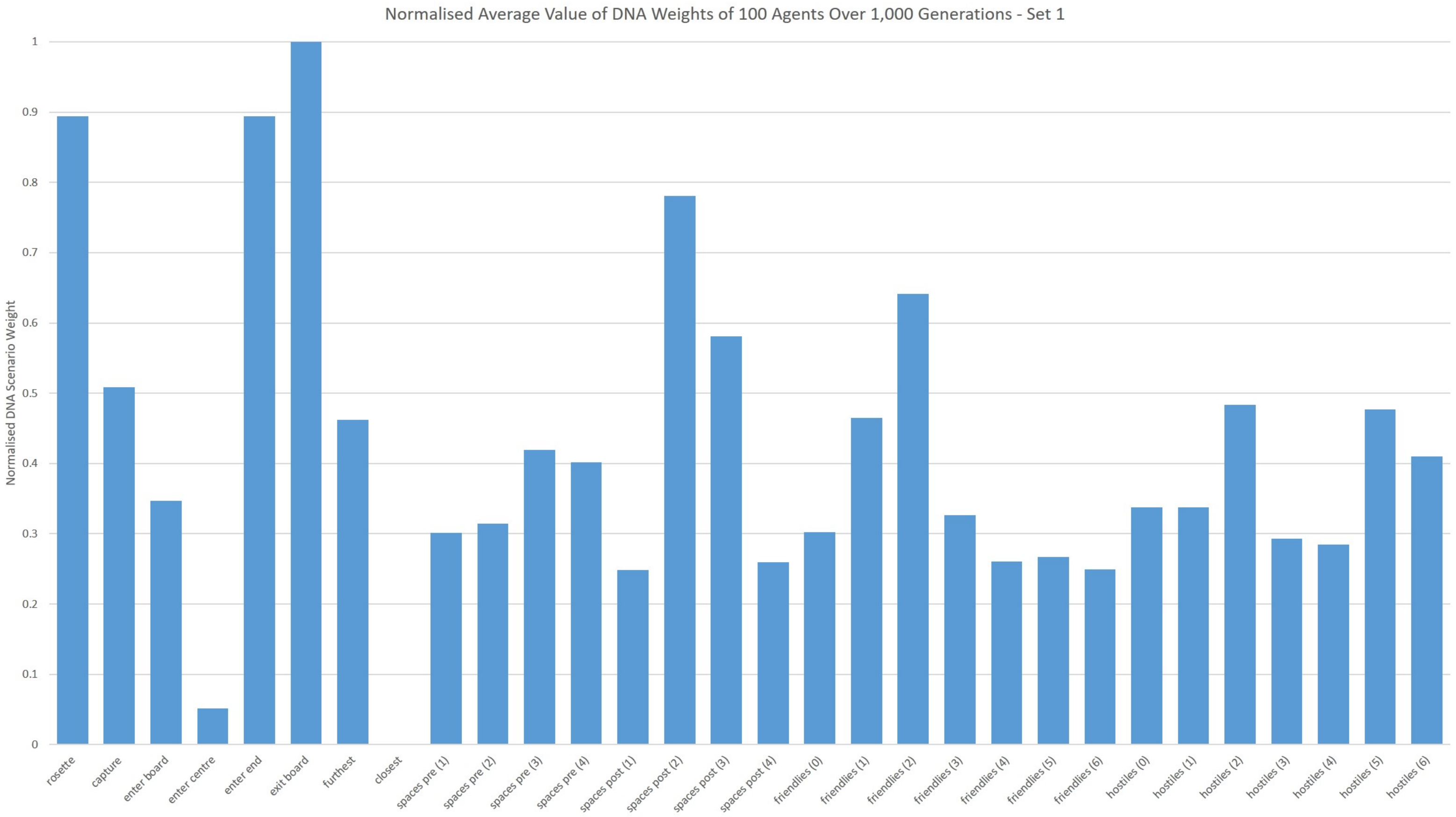


Figure B.5: Normalised average values of DNA scenarios of 100 agents after 1,000 generations - Set 1

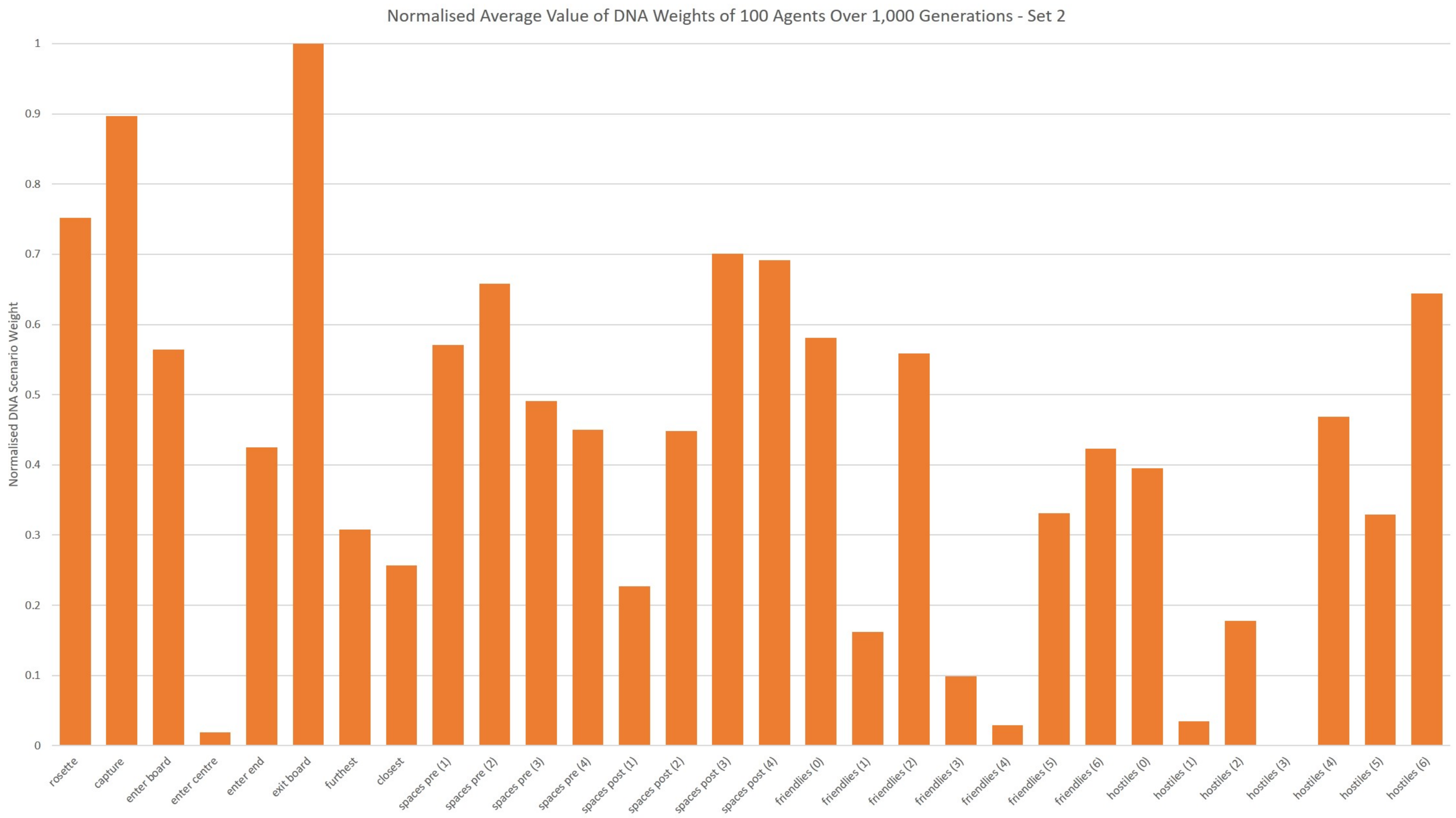


Figure B.6: Normalised average values of DNA scenarios of 100 agents after 1,000 generations - Set 2

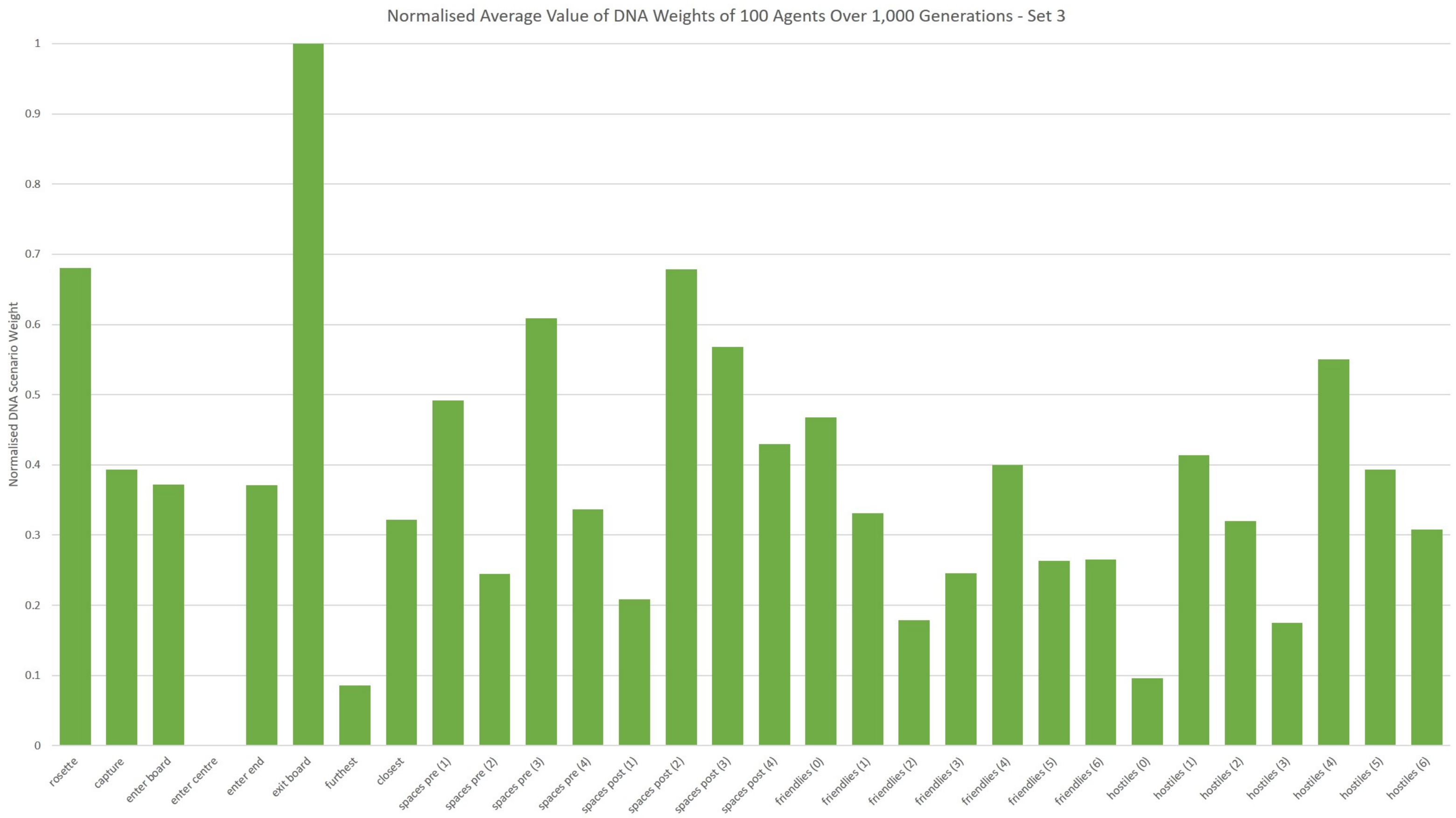


Figure B.7: Normalised average values of DNA scenarios of 100 agents after 1,000 generations - Set 3

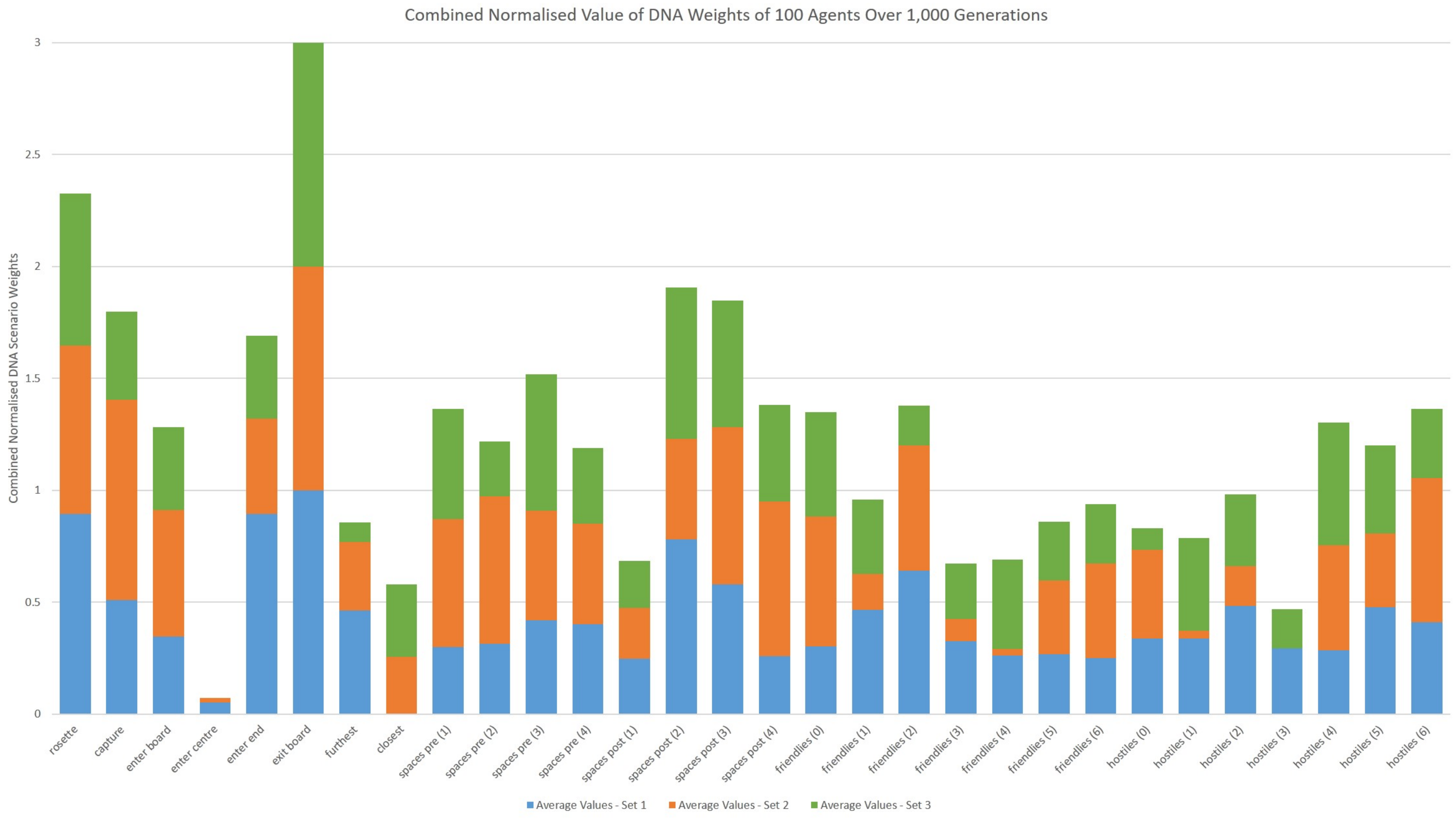


Figure B.8: Normalised average values of DNA scenarios of 100 agents after 1,000 generations (stacked)

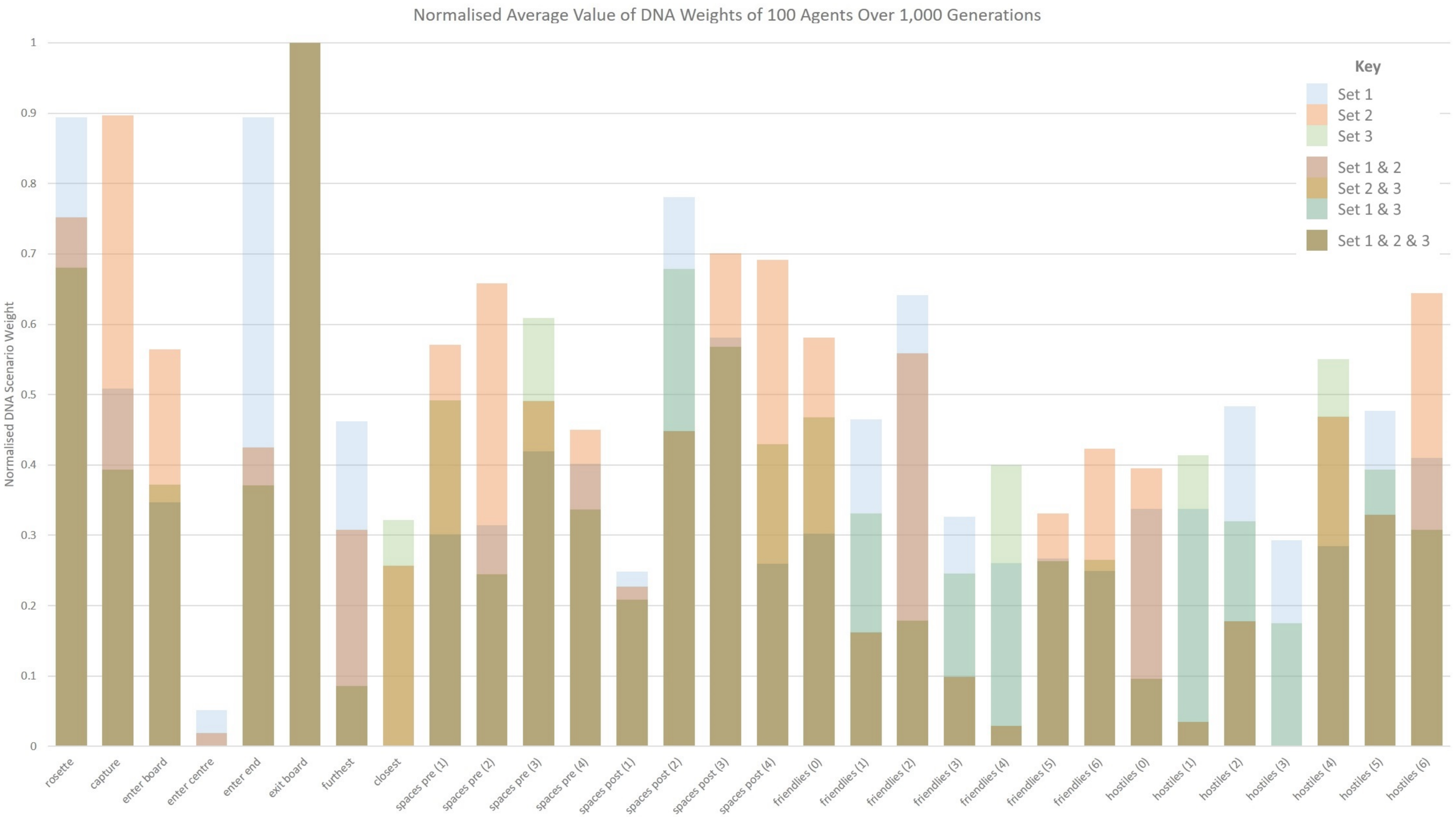


Figure B.9: Normalised average values of DNA scenarios of 100 agents after 1,000 generations (overlaid)