School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCE

Analysis of Artificial Intelligence Techniques in Strategy Games: Hive

Amaan Ibn-Nasar

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2019/2020

40 credits

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	Report	Minerva (05/05/20)
Project Repository	Gitlab repository: `https://gitlab.com/ll16ain/final-year-project`	Supervisor, Assessor (05/05/20)
Software Execution Instructions	Gitlab repository: `https://gitlab.com/ll16ain/final-year-project/-/wikis/Build-instructions`	Supervisor, Assessor (05/05/05)

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) _____

Summary

This project intends to explore Artificial Intelligence techniques and how they can be applied to adversarial games. The techniques researched and implemented include, the Minimax algorithm, Genetic Algorithms and Evolutionary Computation and Reinforcement Learning algorithms. These techniques will then be evaluated within the context of the game Hive. The results show that the techniques improve an agent playing the game but requires further work to increase efficiency and effectiveness of the techniques.

Acknowledgements

I would like to thank my supervisor, Brandon Bennett, for his help and guidance throughout the project.
I would also like to thank my assessor, Sam Wilson, for the feedback given in the progess meeting.

Lastly, I would like to thank Anna Bertram for all the support she has given me throughout this project.

Contents

Chapter 1

Introduction

1.1 Project Context

There have been numerous studies into the creation of Artificial Intelligence (AI) agents to play adversarial games [31, 34, 35]. These AI agents allow for the study of rational decision making and cognitive processes. The purpose of this project is to explore AI techniques and their application in games. These techniques will be evaluated and compared to previous studies to help inform future research in the field of AI and Game Theory. These techniques will be applied to the game Hive. Hive is a game comparable to chess in that it has a simple objective and pieces with a variety of movement options. The rules can be learned easily but learning optimum strategies and when they are applicable can be very difficult.

1.2 Aims

The primary aim of this project is to create and implement an AI algorithm, capable of playing against a human in a game of Hive. The secondary aim is to develop multiple algorithms using different AI techniques and evaluate them against each other to find the most intelligent algorithm. Implementations of AI algorithms in previous studies will be researched before designing and creating an algorithm for the game Hive.

1.3 Objectives and Deliverables

The main objectives of this project are:

1. Develop software allowing users to play the tile-based board game.
2. Implement an algorithm that can simulate a player.
    (a) Develop software allowing for two AI agents to play against each other.
3. Use AI techniques to improve the algorithm and create more algorithms.
4. Evaluate the algorithms and compare them to each other.

1.4 Risk Mitigation

The project creates basic versions of both the game implementation and the AI. This means that if there are issues with the project and it can't be completed in full then the basic implementations can be used, thus reducing the risk of the project.

1.5 Project Timeline

The initial plan (shown in Figure 1.1) shows the development process as it was intended.

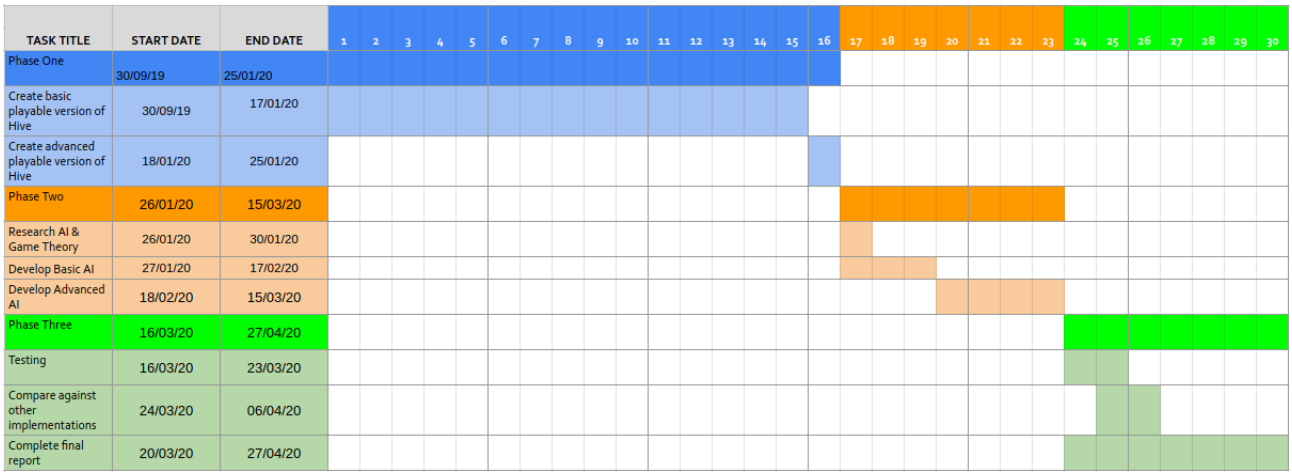


Figure 1.1: Initial Gantt Chart

The initial Gantt chart was useful to plan the overall development process for the project. However, when developing the AI agent and writing the report, it became apparent that the schedule would need adjusting. The amount of time required to write the report and test the solution was underestimated. The revised chart is shown in Figure 1.2.

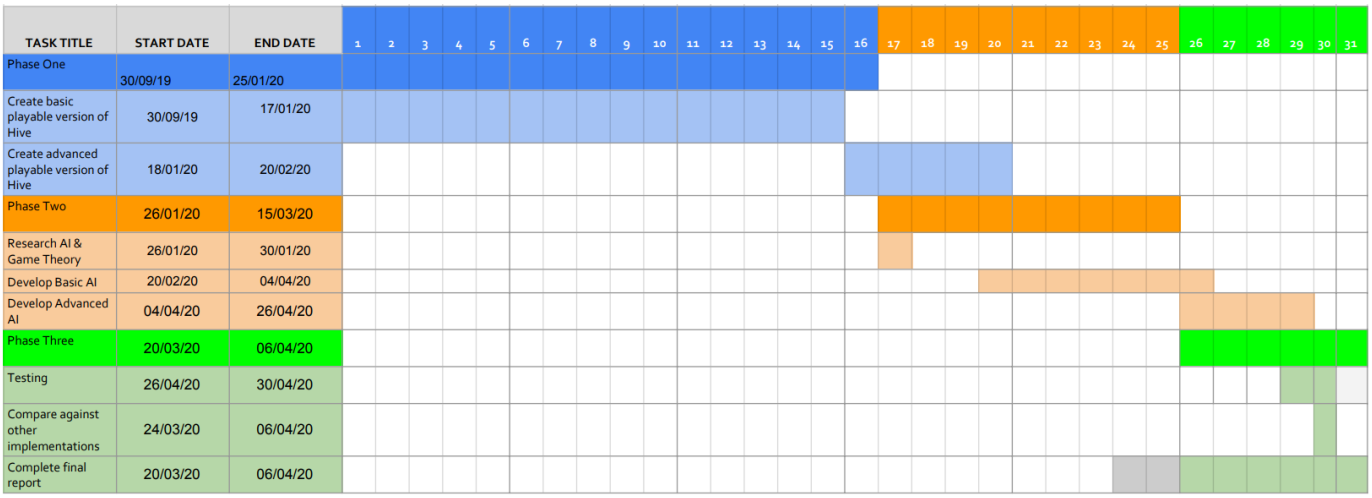


Figure 1.2: Revised Gantt Chart

1.6 Project Methodology

This project was planned and developed using agile methodology. This allowed for an incremental development that compliments exploratory software projects such as this one. It also ensured quick implementation with focus on development and producing a high-quality system [39].

Chapter 2

Background Research

2.1 Game Theory

2.1.1 Introduction to Game Theory

Game Theory analyses behaviour of two or more players in a game with a payoff. A payoff is a value assigned to the outcome of a game. A game is conceptualised as an interaction between multiple players where each player's payoff is affected by decisions of others. The rules of the game are known to every player and it is assumed that all players are rational. A rational player is one that will select the best move possible that aligns with their interests [24]. A player is defined as a combined list of interests. Therefore, this allows for a player to consist of multiple parties, all acting towards the same goal [28].

A strategy is a plan of actions that the player enacts when playing the game. A pure strategy describes the actions of the player in every possible state of the game [9]. A dominant strategy describes the best actions a player can take to produce the highest payoff regardless of any other player's actions [14].

Game Theory has applications in numerous sectors, due to the abstract nature of its definitions, including, economics and business [33], politics [6], biology [37], and computer science and logic [29].

2.1.2 Classifications of Games

Competitive and Cooperative Games

Competitive games are games in which each player is working against one another to achieve their goal. The most prevalent example of a competitive game in the field of Game Theory is the Prisoner's Dilemma. Two prisoner's have been accused of a crime. They are placed in separate rooms so they cannot collude with one another. They have two options, confess or do not confess. If one prisoner confesses then they are released and the other prisoner is sent to prison for 10 years. If both prisoners confess then they are both sent to prison for five years. If both prisoners do not confess then they are sent to prison for three years. This can be illustrated by Table 2.1. The first number in each box represents the payoff for Prisoner One and the second number represents the payoff for Prisoner Two.

		Prisoner One	
		Confess	Do not Confess
Prisoner Two	Confess	(-5, -5)	(-10, 0)
	Do not Confess	(0, -10)	(-3, -3)

Table 2.1: Outcomes of the Prisoner's Dilemma

The best outcome for either prisoner is for one to confess and the other to not confess. However, by analysing the table of outcomes, a simple dominant strategy can be devised.

The strategy dictates that either player should confess. Analysing a game becomes trivial when a dominant strategy exists. Player's do not have to predict what other actions player's may take while still relying on their own best strategy. Using the dominant strategy, predictions can be made that state that both players will confess resulting in both players being sent to prison for five years. When dominant strategies don't exist, analysis can be performed by examining equilibrium [14]. A Nash equilibrium is a set of strategies such that no player can deviate from their strategy and expect a higher payoff [14]. In the Prisoner's Dilemma, the outcome of both prisoner's confessing is a Nash equilibrium. If Prisoner Two confesses then it is more advantageous for Prisoner One to also confess for the payoff of minus five. If Prisoner Two does not confess then it is more advantageous for Prisoner One to confess for the payoff of zero. This equally holds for Prisoner Two's best actions [27]. Examples of competitive games include, Chess, Rock-Paper-Scissors, Hive and Noughts and Crosses. Each player has strictly opposing goals in these games and therefore, they are competitive.

Cooperative games are games in which player's can assemble and dissolve coalitions. In contrast to competitive games, cooperative games address the outcomes that result from players working together. Due to the nature of cooperative games, only games with more than two players are considered. Players form coalitions to secure a bigger payoff than the actions of any one player. Collaborative player actions follow the process of a bargaining strategy where each player is still adhering to maximising their own payoff [24]. This bargaining strategy can be modelled, with player's actions in the bargain represented as moves in a competitive/non-cooperative game [25]. This can then be analysed as a competitive game. Risk is an example of a cooperative game. In Risk, the rules neither allow nor forbid alliances to be formed. Players will often form unofficial coalitions that can be dissolved at any time in the game, thus Risk is a cooperative game.

N-Player Games

Games can have one player, two players or n-players, where n is a number greater than two. These categorisations exist as games with different numbers of players can drastically change the properties and classifications of the game. One player games are usually simpler to analyse due to there only being one entity to account for. Two player games are generally competitive, with players having to consider their opponents possible moves in order to win. N-player games can be competitive or collaborative, and therefore require more consideration when analysing them.

Zero-sum Games

A zero-sum game is a game in which a positive payoff for one player means a negative payoff for another player. The payoffs of all players sum to zero. From the definition of zero-sum games, only games with more than one player can be considered to be zero-sum. Two player zero-sum games are strictly competitive as the player's interests are diametrically opposed [9]. Games in which all players can gain or lose are referred to as non-zero-sum. Any non-zero-sum

game with n-players is strictly equivalent to a zero-sum game with $(n+1)$-players where the $(n+1)$th player portrays the the global gain and loss of the game [42]. Examples of zero-sum games include, Chess, Checkers, Poker and Arm Wrestling, although there are many more. In each case, for every outcome, the overall payoff sums to zero. An example of a non-zero-sum game is the Prisoner's Dilemma.

Simultaneous and Sequential Games

In simultaneous games, no player has information about opposing player's actions before they decide their own action [7]. Consequently, opposing player's actions do not influence the actions of any other player. The Prisoner's Dilemma is an example of a simultaneous game. The most widely known, simple simultaneous game is Rock-Paper-Scissors. Neither player knows the other's action and both make their action at the same time. The outcomes are shown in Table 2.2. The first number in each box represents the payoff for Player One and the second number represents the payoff for Player Two.

		Player One		
		Rock	Paper	Scissors
Player Two	Rock	(0, 0)	(1, -1)	(-1, 1)
	Paper	(-1, 1)	(0, 0)	(1, -1)
	Scissors	(1, -1)	(-1, 1)	(0, 0)

Table 2.2: Outcomes of Rock-Paper-Scissors

The outcomes of Rock-Paper-Scissors are displayed in *Normal Form* representation; most simultaneous games are represented in this way (later discussed in Section 2.1.3). Rock-Paper-Scissors is also an example of a zero-sum game. Simultaneous games are also implicitly *imperfect information* games as each player has no information about other players' actions.

In sequential games, players decide their turns consecutively. This allows for a player to use the additional information of the previous player's action to inform their decision [7]. An example of a sequential game is Noughts and Crosses. Each player decides their turn after the other. A simplified diagram of outcomes is shown in Figure 2.1. The outcomes of Noughts and Crosses are displayed in *Extensive Form* representation; most sequential games are represented in this way (later discussed in Section 2.1.3). The actions each player can take are represented by two letter commands. For example, *MC*, represents the cell on the middle row in the centre column.

Perfect and Imperfect Information Games

Games have perfect information if, for any player making a move in any state of the game, that player has information of all the previous moves that occurred to result in that state [17]. Chess is a game with perfect information. Games with sequential turns, no hidden information and an element of chance, for example Monopoly, are sometimes thought of as having perfect information. Imperfect information games have information that is hidden from a player. Card games in which players hide their hands from other players are examples of imperfect information games.

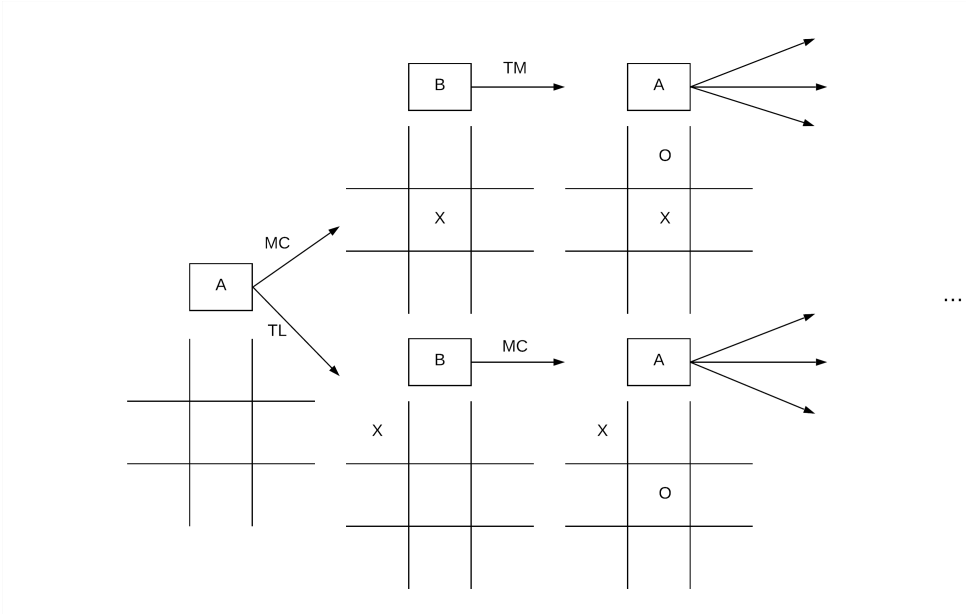


Figure 2.1: Simplified outcomes for Noughts and Crosses

2.1.3 Game Representation

Normal Form Representation

Normal Form represents games as a matrix. Normal form representation includes all possible strategies and payoffs for each player. For example, in a two player game, each column will represent an action for Player One and each row will represent an action for Player Two. Each element in the matrix will correspond to the payoff for the actions of Player One and Player Two [20]. Examples of games represented in normal form are the Prisoner's Dilemma (Table 2.1) and Rock-Paper-Scissors (Table 2.2). Simultaneous games are represented in normal form as all players' actions are decided at the same time and without knowledge of other players' actions.

Extensive Form Representation

Extensive Form represents games as a decision tree. The choices players' can make are depicted by the nodes and actions the players' can take by the edges. The leaves of the tree represent the terminating states of the game and the corresponding payoff for each player. Extensive form allows for the representation of sequential moves where players can only decide on an action after another player has taken their turn [20]. An example of a game represented in extensive form is Noughts and Crosses (Figure 2.1). Both simultaneous and sequential games can be represented by extensive form, however, for simultaneous games, normal form is preferred. An advantage of extensive form representation is that the flow of a game can clearly be seen. It is also trivial to see the history of a game, given a state, as the tree can be traversed to the root from that state. When utilised for AI techniques, an agent will usually have a *local extensive form* representation available to it. This is a subsection of the full game tree consisting of the current state and the possible next states.

2.2 Artificial Intelligence Techniques

2.2.1 What is Artificial Intelligence?

There are four main approaches to consider when discussing and defining AI [30].

Thinking Humanly

This addresses the processes traditionally related to human thinking and their automation [3]. It focuses on modelling the human mind and identifying how humans think. This can be done through the use of introspection, psychological experiments and neuroimaging. Introspection researches one's own thoughts. Psychological experiments measure a person's behaviour in a particular scenario. Neuroimaging observes the brain of a person in a specified situation [30].

Acting Humanly

This approach defines AI as the creation of machines that execute the same functions that humans can naturally perform [18]. This is also known as the Turing Test approach. The test designs a scenario in which a human interrogator proposes questions to a computer. The computer passes the test if the interrogator, when receiving the responses, cannot tell that the answers have been written by a machine [41]. For the computer to pass the test it must be able to process language naturally, communicate successfully and store knowledge. It must be able to use the stored information to answer questions and draw inferences, and learn to adjust to new situations. However, the necessity of the Turing Test has been challenged with multiple studies claiming it to be the terminal goal in the field of AI, while others cite it as damaging to the area of study [32]. Humans are considered to be the most intelligent creatures and games have a great impact on this. Furthermore, for hundreds of years Grandmasters of Chess were thought to be the most intelligent humans until computers could reliably beat the best Chess players [8, 26]. Therefore, acting humanly does not necessarily mean to act intelligently.

Thinking Rationally

This approach takes a philosophical approach to define AI. It studies the logic and calculations that allow entities to comprehend situations, rationalise and take action [43]. It examines the laws of thought and their function in the operation of the mind. Logic allows for an AI to make inferences based on existing knowledge. For example, if a system knows that Dave is a human and all humans are mammals, it can infer that Dave is a mammal. In theory, programs can be written that can solve any problem written in logical notation by the use of inference and reasoning and then extend those programs to create intelligent systems. Issues then arise concerning the transformation of informal knowledge into logical notation. If this is achieved, the subsequent step is to create a system that can take the logical notation and produce a solution. This can be difficult for problems with numerous facts, with even a few hundred exhausting the resources of most computers without guidance as to a starting point [30].

Acting Rationally

This approach investigates AI as a rational agent. An agent is simply an entity that acts. Ra-

tional, in this context, means an agent that takes actions to maximise some function and attain the best outcome. Rational agents will think rationally, but will also face situations in which there does not exist a provably right action to take, and must still make a decision [30]. Game playing can be used to test the rationality of an agent and how effective it is.

The distinction between human and rational behaviour does not imply that humans are irrational in terms of derangement or emotional instability. Instead, it suggests that humans are not perfect and make mistakes [30].

2.2.2 Uninformed Search Strategies

A game tree is a data structure that stores all possible actions in a game from the first to the final moves. The root of the tree is the initial state of a game and the children of the root node are actions that can be taken from that initial state. All games that can be represented in *extensive form* have a game tree. If an AI agent has access to the game tree and is aware of the current state the game is in, then the goal of winning the game becomes a search problem. The agent can search the tree for a state where it wins, known as a goal state. In uninformed search strategies, the only information known about the states is whether it is a goal state or not [30]. Given infinite computational resources, uninformed search strategies will always find a goal state. In practice, uninformed search strategies are not commonly used for complex problems as the resources required are too high when compared to other search techniques. The difference between uninformed search strategies is in how they search the game tree.

Breadth-First Search

Breadth-first search (BFS) explores the game tree, starting with the root node and then its children. It checks all child nodes and then searches their children [16]. In Figure 2.2 an example of BFS is shown. The red coloured nodes represent nodes currently being explored and the green nodes represent nodes that have been explored. The nodes in grey represent unexplored nodes.

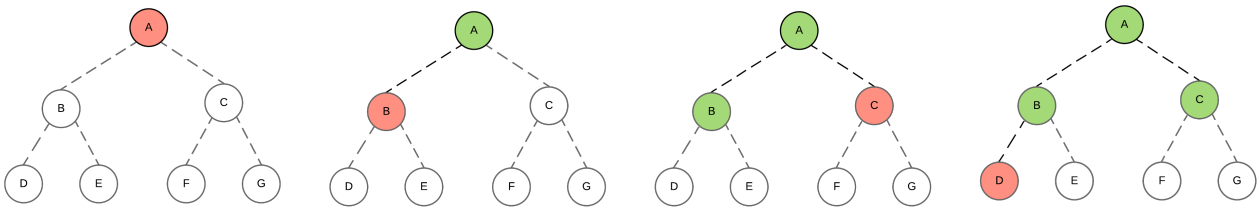


Figure 2.2: Breadth-First Search example

Depth-First Search

Depth-first search (DFS) explores the game tree similarly to breadth-first search. However, it explores each child's children before exploring its siblings [16]. An example is shown in Figure

2.3. The nodes are coloured similarly to Figure 2.2.

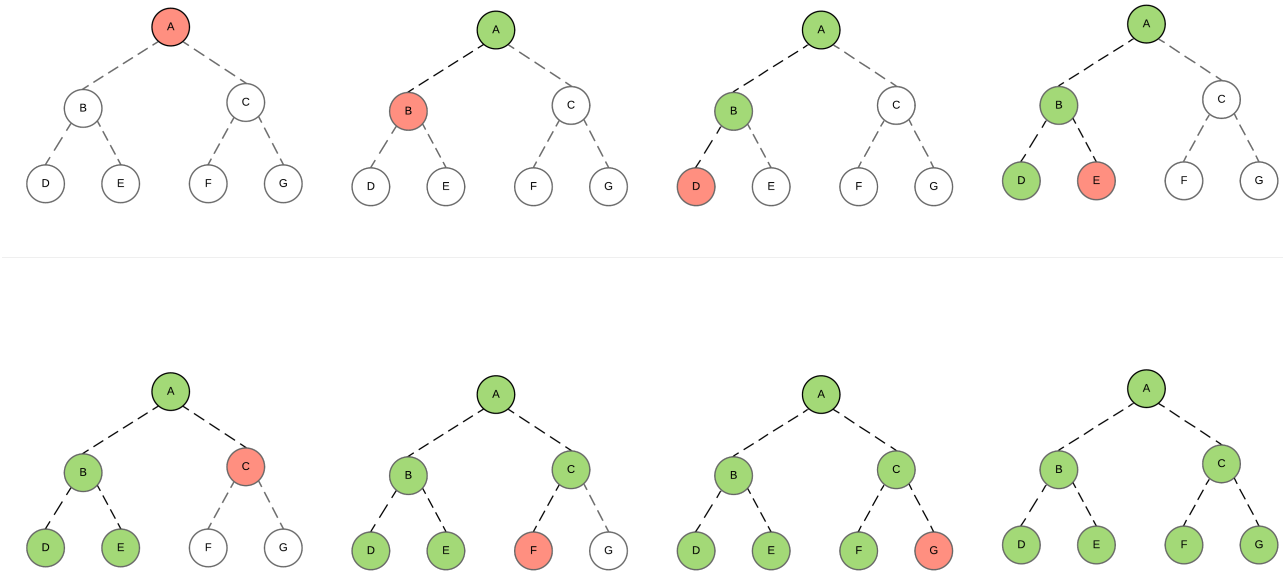


Figure 2.3: Depth-First Search example

2.2.3 Informed Search Strategies

Informed search strategies use additional information about the problem to find solutions more effectively. This is known as the *best-first search* approach, in which exploration of nodes is chosen through the use of an evaluation function [30].

Heuristic Evaluation

A heuristic is guidance to follow to solve a problem, although it may not guarantee success. For example, in Checkers, to win, the player may try to capture an opponent's piece at every turn. However, this may not be optimal as moves that allow for more strategic positioning of one's own pieces may be overlooked. For AI agents, heuristic evaluation functions are devised. A heuristic evaluation function takes a state of the game as its input and outputs a numerical value to represent that state. The evaluation function is usually denoted by *f(n)* and the heuristics used in the evaluation function by *h(n)*. These functions can then be used to inform the agent of the next best move to make. This drastically reduces computation required to solve the problem as the agent searches for the goal with the added information provided by the evaluation function. This allows for a narrower search than BFS and DFS as not all of the branches of the game tree are explored, only the branches with the greatest chance of success [15].

The greedy best-first search algorithm evaluates states with $f(n)=h(n)$, meaning that the evaluation function assesses states only using the heuristic function. This search algorithm explores nodes trying to achieve the goal as quickly as possible.

A* search is another best-first algorithm. It considers an additional function *g(n)* that denotes the cost to reach a node of the game tree. The evaluation function combines this cost function with the heuristic function to give *f(n)=g(n)+h(n)*. A* explores nodes in increasing order of cost and therefore never overestimates the total cost of a solution [15]. An admissible heuristic is one that does not overestimate the cost to reach the goal state.

Heuristic functions can be constructed from a set of features of a game. These features are countable components of the game that are then combined with a set of weights. The weights can be increased or decreased to adjust the value of a feature and subsequently how much that feature is considered overall in the function. This is a weighted linear function, the sum of which gives the heuristic value for a state. For example, in Chess, features could be pieces the player has and the weights would reflect the importance of certain pieces, with queen and rook being favoured above a pawn. Heuristic functions can also be constructed from simpler versions of a game with fewer restrictions. The simpler games have more easily solvable solutions and can be used as a heuristic in the original, more constrained game [30].

2.2.4 Adversarial Search

Adversarial search concerns search algorithms in which two opposing agents explore the same search space for a solution. As a result of this, adversarial search algorithms commonly alternate between players when finding a solution.

Minimax

The minimax algorithm is used in two player, sequential, perfect information, competitive games, and assumes that both players will play optimally. It represents one player as *MAX* for maximising the payoff of the game and the other as *MIN* for minimising the payoff of the game. The algorithm searches the game tree to find the best action a player can take, given that the other player plays optimally. The algorithm takes each branch of the game tree and recursively explores deeper until it reaches a goal state or until it reaches a predetermined maximum depth. The payoff of the state is then evaluated and returned. It then alternates, assigning the parent node with the minimum and maximum payoff of its children, until it reaches the root of the tree. This imitates both players, maximising and minimising, deciding the best move at each turn and thus, returns the best action that the player can make.

In simple games with few actions the algorithm can execute quickly. However, in more complex games the number of actions a player can make can cause an exponential increase in states that need to be evaluated. The number of states created at each level of the game tree is called the branching factor. A branching factor of 10 means that from each state there are 10 actions that can be taken that result in 10 different states. At the root level of the tree there will be 10 states but at the next level there will be 100 states; Each state from the previous level will create 10 states for the next level. Techniques can be used to reduce the number of branches that need to be explored, such as a *depth limit* and *Alpha-Beta pruning*. A depth limit stops the algorithm from searching once it has reached a certain level of the tree [5]. A depth limit can

be decided by accounting for the branching factor and the resources needed to explore each level of the tree. Different games may require different depth limits in order to search the game tree effectively. If a depth limit is too low, agents may not make good decisions. This is known as the *Horizon Effect* [4].

Alpha-Beta Pruning

Alpha-Beta pruning is a method used to reduce the computation time of the minimax algorithm. It reduces the number of branches that need to be explored by the algorithm. Two components are introduced, *alpha* and *beta*, which are used to determine which branches need not be explored. Alpha represents the minimum score that the maximising player can obtain and beta represents the maximum score that the minimising player can obtain. Branches can be pruned when beta is less than alpha as the minimising player will always deny the option to choose that node. This reduces the time taken for the minimax algorithm to explore the tree drastically as branches of the tree that can never be reached, due to optimal moves by each player, are not explored [31].

2.2.5 Evolutionary Computation

Evolutionary computation techniques take the principles of evolution from biological evolutionary theory and applies them to algorithms in order to search for solutions where other techniques may not be applicable, or yield unsatisfactory results. Contrary to uninformed and informed search strategies, that either exhaustively search or search one node at a time, evolutionary search algorithms directly search. This is because the algorithms are population-based and search by adapting consecutive generations of entities [36]. Evolutionary computing can be used to find solutions to problems where no obvious solutions present themselves. This is because it constantly adapts with each generation and produces a better solution.

Genetic Algorithms

Genetic algorithms (GAs) follow the principle of natural selection. GAs search by taking an initial population, consisting of random solutions called chromosomes. This population is then applied to the problem and a fitness function is used to determine the varying levels of success of each chromosome. Then, a new generation of chromosomes are computed by selecting and evolving the best chromosomes of the previous generation. The new chromosomes are called offspring. The evolution procedure to generate offspring is carried out through two operations. The first is combining two chromosomes using a crossover operation. The second is transforming a chromosome with a mutation operation. The new generation must have the same population size as the previous, so new offspring are generated by selecting parent chromosomes from the previous generation according to their fitness function value. The fitter a chromosome is, the higher the chance of being selected [10].

Chromosomes are usually encoded as fixed-length binary strings with each bit in the string corresponding to a characteristic of the chromosome. A simple example of this is shown when taking an animal, such as a cat, and encoding them. The string could contain ones for bits that correspond to traits of a cat, "meows", "chases mice", "predatory", and zeroes for bits

corresponding to traits a cat does not have, "drives", "writes", "talks". The crossover operation takes two binary string chromosomes as input and a crossover point is determined. The chromosomes produced are the same as the first input until the crossover point. From the crossover point, the bits in the string are exchanged with the other input. The mutation operation alters a chromosome by *flipping* a bit in the string. A zero will be inverted to become one and vice versa [13]. Chromosomes can be represented in other formats but as a result of this, the mutation and crossover operations must be specialised to the new representation [22].

When creating a new encoding for chromosomes, the encoding must be evaluated to ensure it is valid [10]. The evaluation criteria are:

- Space - Chromosomes must require an appropriate amount of memory;
- Time - Evaluating and performing operations on chromosomes must have small time complexities;
- Feasibility - All chromosomes produced by operations must correspond to feasible solutions;
- Uniqueness - A one-to-one mapping when decoding a chromosome to a solution is desired;
- Heritability - The crossover operation must produce an offspring which represents the solution of combining its parents;
- Locality - The mutation operation must produce an offspring which represents a solution similar to its parent;

2.2.6 Machine Learning

Machine learning studies algorithms that can improve through experience. These algorithms typically require a dataset to learn from and construct a model that can then be used to make decisions on new data. The general machine learning process is described as follows. Data, concerning the problem to be solved, is collected to form a dataset which can be learned from. Features that are most useful to the problem are then identified. An appropriate machine learning algorithm is chosen and trained using the dataset. The resulting model is then evaluated using suitable metrics to determine the overall validity of the model to solve the problem. Every machine learning algorithm can be classified by the type of learning it follows. There are generally four classifications, supervised, unsupervised, reinforcement and evolutionary learning (discussed in Section 2.2.5) [29].

Supervised Learning

In supervised learning, algorithms are provided with input data as well as target data. Learning occurs by comparing the output of the algorithm with the target data. The parameters of the algorithm are then adjusted to produce better results. Supervised learning algorithms are used for classification and regression problems. Classification problems take a set of inputs and must categorise each input according to predetermined classes. Each output is strictly classified as one value. Regression problems are similar to classification but are used to approximate, as

the outputs may take any value within a range. Algorithms that can be used to implement supervised learning include support vector machines, decision trees and neural networks [2, 29].

Unsupervised Learning

Unsupervised learning algorithms are similar to supervised learning algorithms but they do not have target data. As a result of this, unsupervised learning algorithms cannot be used for regression problems. For classification problems, unsupervised learning algorithms aim to learn and identify common features in inputs to classify them. Most commonly, clustering methods are used in unsupervised learning algorithms. An example of a clustering algorithm is k-means [2, 23].

Reinforcement Learning

Reinforcement learning uses techniques from both supervised and unsupervised learning. The agent is given information to confirm if their answer is correct but is not given information on how to improve. The agent must explore to see which actions result in correct answers. Reinforcement learning algorithms aim to maximise reward, where reward is feedback that informs the algorithm if its actions were beneficial or detrimental. Therefore, the algorithm explores its environment and receives a reward as feedback. This trial-and-error approach to searching the environment is a key aspect of reinforcement learning. However, actions taken may influence potential future reward, as well as the immediate reward from taking that action. This concept is called *delayed reward* [2, 40].

To create a reinforcement learning system the environment must be modelled and a reward function, value function, and policy devised. The environment is usually represented as a *Markov Decision Process* (MDP). An MDP can be written in the form $<S,A,T,r>$. S is the set of states of the environment. A is the set of actions that an agent can make. T is a transition function that describes which state an agent arrives in, given an initial state and action. Lastly, r is a reward function that describes the reward an agent will receive given a state and action. A policy, π, describes the action an agent will take in a given state. The value function describes the *expected reward* an agent can receive given a state while following the policy. The value function *discounts* further rewards meaning that, the further ahead it looks, the less the reward is worth. The system described thus far models an environment and can plan to maximise reward but it does not learn and is invalid in situations where a model of the environment is not available [21].

Temporal Difference Methods

Temporal difference (TD) methods update the value function at each state by examining the next action and its value. This means that the algorithm learns a new policy, based on approximations of values of the next state, and the actions it has taken in the environment. Q-Learning is a TD method algorithm that learns about the *optimal policy*. The optimal policy is the policy describing the best action to take in all states. Q-Learning always updates the policy with the value of the best action. It does not necessarily take the action from the policy, so it is defined as an *off-policy* algorithm. Sarsa is a TD method algorithm that also learns the optimal policy,

but it follows the current policy for actions it makes. This is known as an *on-policy* algorithm. Q-Learning and Sarsa differ in the updating of their policies. Q-Learning will always update the policy according to the best action it can take in the next state, whereas Sarsa will update the policy according to the best action it can take while still following the policy [21].

Passive Learning

A type of reinforcement learning is passive learning. Passive learning uses reinforcement learning techniques to adapt a set of heuristics so that an agent can perform better. It is a technique with applications in game playing AI. A weighted linear function is used to evaluate the states of a game (discussed in Section 2.2.3). The aim of the technique is to learn the most appropriate weighting of each feature. This is achieved by running iterations of the game with each iteration using a slightly altered set of weights. Each set of weights differs by only one weight which has a different value. The learning takes place once all iterations have concluded. The set of weights with the most successful outcome is then used as the basis for the next round of iterations [30, 31].

2.3 Hive - Board Game

The board game Hive is much like Chess. It has a simple premise and the rules are easy to learn but very difficult to master. It is a two player strategy game with a unique concept in that it is a board game with no explicit board. The board is whatever surface you would like to play it on [11].

2.3.1 Classification

Hive can be classified, using game classifications described in Section 2.1.2, as a two player, competitive, sequential, perfect information, zero-sum game. The game cannot be played with a single player or more than two players. Both players in the game are diametrically opposed in their goals with no possibility for cooperation. Each turn is taken sequentially, with each player possessing information of the opposing player's previous turn. No elements of the game are kept hidden from either player. Each instance of a winning game has exactly one winner and one loser. This can be represented as a payoff of positive one for the winning player and a payoff of negative one for the losing player. In cases of a draw, both players have a payoff of zero.

2.3.2 Composition

The game consists of 22 hexagonal pieces, each with one of five bug designs on them. Each bug has different movement options.
Each player has 11 pieces including:

- One Queen Bee
- Two Spiders
- Two Beetles
- Three Grasshoppers
- Three Soldier Ants

This project will solely consider the original edition of the game, despite other available editions containing expansions with more pieces and movement options.

2.3.3 Gameplay

The goal of the game is to completely surround your opponent's Queen Bee with pieces. The surrounding pieces may belong to either player. Once the Queen Bee is set, a player can either move a piece or play a piece. A draw is reached if the final move surrounds both players' Queen Bee. A draw can also be agreed when a stalemate has been reached with no winning move being possible.

2.3.4 The Rules

General Rules

The most fundamental rule is the One Hive rule. This means all pieces must be connected. Thus, any game state can be represented as a connected graph with the pieces being the nodes and the edges connecting adjacent pieces. A move cannot be made if it would disconnect the hive and then reconnect it. Figure 2.4 shows the One Hive rule.

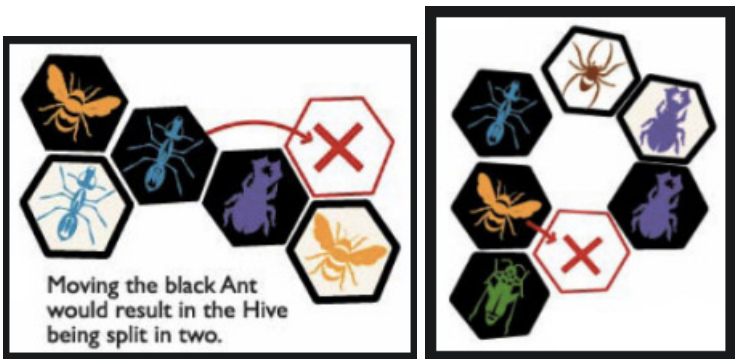


Figure 2.4: One Hive Rule [1]

When placing a piece, it must only be in contact with pieces belonging to that player, with the exception of each player's first tile (shown in Figure 2.5).

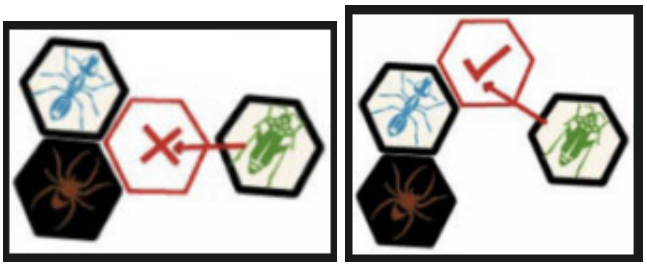


Figure 2.5: Placing Rule [1]

The Queen Bee must be placed within four turns. Once the Queen Bee has been placed, the player can decide whether to move a piece or set a piece. Once a piece is in play it cannot be removed. If a player has no actions available to them, the turn passes to the opponent.
When moving a piece, it must not disrupt any other game pieces, meaning it has the ability to

slide. If a piece cannot physically slide from its position then it cannot move (shown in Figure 2.6). Exceptions to this rule include the Grasshopper and the Beetle due to their movement facilities (later discussed in Creature Specific Rules).

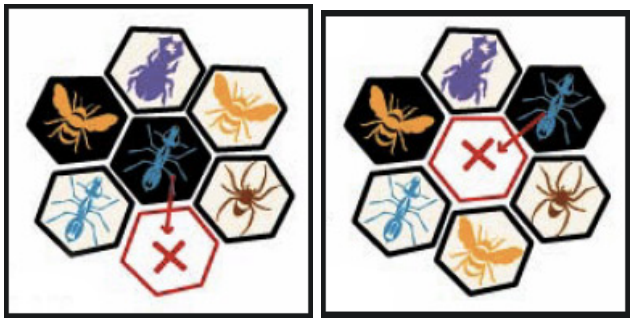
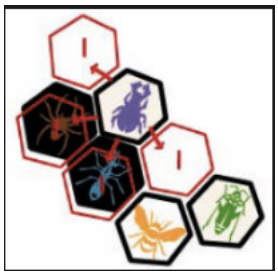


Figure 2.6: Sliding Rule [1]

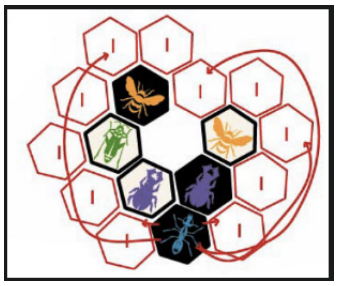
Creature Specific Rules

Queen Bee: Each turn, the Queen Bee can move one space in any direction, as long as the general rules are enforced.
Beetle: The Beetle, like the Queen Bee, can move one space each turn. It can also move over other pieces. If the Beetle is on top of another piece, the piece below it cannot move. When first playing a Beetle it cannot be placed on top of another piece.
Soldier Ant: The Soldier Ant can move any number of spaces around the perimeter of the hive, as long as the general rules are enforced.
Spider: The Spider, like the Soldier Ant, moves around the perimeter of the hive. However, it moves exactly three spaces and it must move in a direct path. For example, it cannot move one space forward and then two spaces backwards. Additionally, it must only move around pieces it has direct contact with, it cannot jump gaps in the hive.
Grasshopper: The Grasshopper jumps over the hive. It can only jump in a straight line and cannot jump over a gap in the hive.

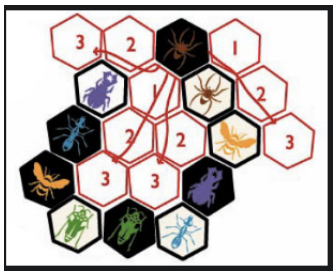
Diagrams of all creature movement options are shown in Figure 2.7.



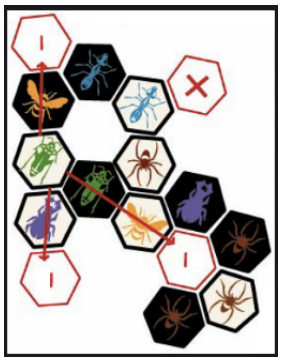
(a) Beetle Movement



(b) Soldier Ant Movement



(c) Spider Movement



(d) Grasshopper Movement

Figure 2.7: Movement Options [1]

Chapter 3

Implementation

3.1 Language and Library Choices

This project used the Python programming language and TkInter library. Python was chosen as it has a simpler syntax than languages such as Java. It also has easy to use Graphical User Interface (GUI) libraries. This would allow for the game to be graphically represented as the presentation of hexagons would be difficult if limited to a text-based interface. TkInter was chosen as it is the most popular GUI library for Python, and is well documented.

3.2 System Overview

The system is comprised of seven main components. Different components are used for different modes of execution. The main components are shown in Figure 3.1.

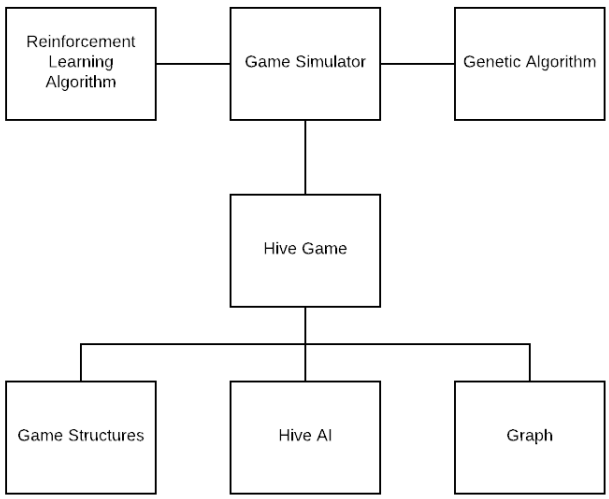


Figure 3.1: Main Components of the System

Hive Game is the component that implements the game. It uses the Game Structures and Graph components in every mode of execution. Both of these components implement key data structures for the game. The Hive AI component is used in modes of execution that require the use of an AI player. The Game Simulator component allows the emulation of multiple games and uses the Genetic Algorithm and Reinforcement Learning Algorithm components to do this. This is used to train an AI player to learn the optimum heuristic parameters. The Game Simulator can also be used to run the game multiple times without training an AI player. This was mainly used to test different configurations of heuristic parameters (discussed in Chapter 4).

3.3 Game Implementation

Before an AI agent could be created, a system to play the game had to be developed. The game was created to allow for three main modes of execution.

1. Two Human Players
2. One Human and One AI Player
3. Two AI Players

The first goal was to create a game where two humans could play. Once this had been implemented, an AI player could be developed to interact with the system and make decisions.

3.3.1 Board Implementation

The physical board game Hive does not require a board. However, the software implementation does, as the pieces must be displayed and stored in some form. Using TkInter's canvas widget proved to be the easiest way to represent hexagons graphically using the *create_polygon* function. The *create_polygon* function generates shapes from a list of co-ordinates representing each point of the polygon. Therefore, to create a hexagon using the function, all of the points of the hexagon must be known. The only information needed when creating a hexagon was the origin and the length of its edges. Using this information, the other points of the hexagon could be calculated, as shown in Figure 3.2. Point A's x co-ordinate was the Origin's x co-ordinate minus 0.5 times the edge length. Point A's y co-ordinate was the Origin's y co-ordinate minus the edge length. Similar calculations were used to generate the other points. On a TkInter canvas, the origin

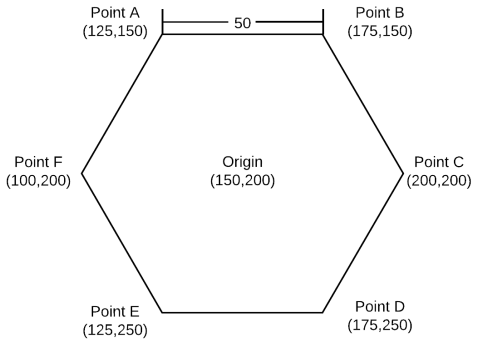


Figure 3.2: A hexagon with all of its co-ordinate points

resides in the top left and increases down the y-axis. The simplest way to graphically represent the board was to create a hexagon based grid. This grid was created by iterating through the number of columns and rows of the board, and creating a hexagon, using the aforementioned method, on the canvas widget at that cell.

The board was now graphically represented but had to also allow for hexagons to be individually tracked. This was achieved through the use of TkInter's tags. Although TkInter creates a unique id associated with any new object created, tagging offered more control over objects and allowed

for easier object location. Tags can be attached to TkInter objects and used to track and manipulate the objects they are attached to. When creating a hexagon, a unique tag was attached to it. The tags maintained individuality by using a counter and adding the prefix "Hex". A class, *Hexagon*, was then designed to store the hexagon data. The class had attributes for the hexagon name and each point of the hexagon. Two more attributes were added for representation of game pieces on the board (discussed in Section 3.3.2). The hexagon name was the same as the tag added to each hexagon on the canvas. A hexagon object was created for each hexagon on the canvas and added to a list of all hexagons to keep track of them outside of events (discussed in Section 3.3.5).

3.3.2 Game Piece Implementation

Game pieces needed an image to represent them. This functionality was implemented through the use of the TkInter canvas' *create_ image* function. When creating the game board (discussed in Section 3.3.1) a canvas image object was also created at the origin of each hexagon. Unique tags for the images were then created and added, similarly to the unique tags for hexagons. As alluded to previously, the *Hexagon* class had two attributes used for representing game pieces on the board. These attributes were *ImageName* and *FileName*. The *ImageName* attribute was the same as the unique tag for the image objects and *FileName* stored the filename for the image that was being represented. A hexagon object could now represent a game piece's position on the board. However, a game piece required more information to be represented accurately. The additional items of information needed were the type of piece and if another piece was on top of it, henceforth referred to as *piece covering*. The type of piece was needed to determine the movement options available (movement of specific pieces discussed in Section 2.3.4). A piece being covered was necessary knowledge as covered pieces could not move. To store the information essential for representing a game piece, a new class was created, *GamePiece*. The *GamePiece* class had attributes to store the position of a piece, the piece type, and if a piece was covering it.

3.3.3 Rule Implementation

One Hive Rule

The One Hive rule was the most fundamental rule to implement. The hive itself could be represented as a graph with nodes representing the pieces and adjacent pieces having an edge between them. A *Graph* class was created to implement this. A graph structure can be represented in code as a dictionary. Each node is a key in the dictionary and the value for a key is a list of all nodes it shares an edge with. The *Graph* class had an attribute that stored dictionary data and an attribute that stored the nodes. The class also had a function to create the dictionary from nodes. The function *createGraph* looped through each node, while ignoring pieces that were being covered, and then looped through all remaining nodes. It then checked the distance between each node and if they were one space away then the the node was appended to the list of adjacent nodes.

The simplest way to ensure the One Hive rule was adhered to, was to not allow any pieces to disconnect from the hive. When placing a piece, only spaces that were adjacent to the hive

were available for selection. This ensured that no pieces, when added to the game, would be disconnected from the hive. This was achieved by using a function *findSurround* which checked the surrounding spaces given an initial space and the type of space to look for. The functionality responsible for placing a piece is described in Section 3.3.4. When moving a piece, no moves that would disconnect the hive were allowed. This was accomplished in the *findMoves* function by checking, via the *findSurround* function, if a potential space to move to had any pieces surrounding it. If it did not, the move was discarded. The two situations when a piece could disconnect from the hive arose when a piece was first placed and when it moved. By adding functionality that prevented disconnections at these two events, it was guaranteed that a piece could not disconnect from the hive and the One Hive rule was enforced.

Placement Rule

The placement rules were implemented by inspecting the pieces on the board when placing a new piece. Only spaces adjacent to pieces of the same colour were made selectable. If a space would be adjacent to two pieces of differing colours then it was removed. This information was attained through the use of the *findSurround* function. A check was made to determine if there was only one piece on the board. If this was true then all spaces around the piece were made selectable. This adhered to the case where the piece being set was the first piece to be placed from either player. The Queen Bee placement rule was implemented by performing a check each turn to detect whether the Queen Bee had been placed. If it was not placed within the four turns then the player was notified and forced to place the Queen Bee.

Sliding Rule

The sliding rule was implemented by checking the number of pieces surrounding a potential move. If the number was greater than four then the move could be removed, as a piece could not slide into it without disrupting other pieces. This ensured no moves were allowed that did not adhere to this rule.

3.3.4 Piece Placement Implementation

To place a piece, each player had to have an inventory from which they could be selected. The inventories for each player were implemented by using TkInter canvas objects and TkInter's layout management to show and hide the relevant inventory. When a player selected a piece from their inventory, the pieces on the board were searched. Using the *findSurround* function, the spaces available to place the piece were then highlighted. When a player clicked on a space the event handler for movement was invoked (discussed in Section 3.3.5). The event handler rendered the game piece on the board, updated key data structures and then deleted the piece from the inventory.

3.3.5 Movement Implementation

The first step to implement movement that adhered to the rules of the game was to allow any piece to move to any space and then impose restrictions for different piece types. TkInter uses an event system to track user input on objects and to call the functions tasked with handling

the event. Events are managed by *event handlers*, which are functions that get called when an event occurs. All *event handler* functions receive *event data* which contains information about the event. Events can be *bound* to any TkInter object by its tag or id.

Game Piece Selection

The first step in moving was selection. A piece had to be selected and then the possible moves it could make could be calculated. A new event handler was created which was called when a piece was clicked. This handler used event data to determine which piece had been clicked and stored it in a global variable. The global variable stored information about a game piece and if it had been selected. This could then be referenced when the event handler managing the movement was called. The last step of selection was to bind spaces that the piece could move to, with the event for movement.

Game Piece Movement

A new event handler was created to manage the movement of a piece. The game piece that was selected was first retrieved from the global variable. Then, the space being moved to was identified from the event data. The movement of the piece itself consisted of two parts, moving the piece graphically and moving the piece logically. To move the piece graphically, the image on the space the piece occupied was removed. The image was then rendered on the new space. To move the piece logically, the position of the game piece was updated in all structures where it was referenced.

Creature Movement

The first creature's movement to be implemented was the Soldier Ant. To enforce the rules regarding the Soldier Ant's movement (described in Section 2.3.4), the moves calculated had to correspond to the perimeter around the hive. This was achieved by exhaustively finding all spaces around all pieces on the board using the *findSurround* function and adding them to a set of moves. Each move in the set was then checked to see if it would break any of the rules.

The Spider's movement rules were similar to the Soldier Ant's. However, it had the added restriction of only moving three spaces. Therefore, the Soldier Ant's moves could be calculated and then any moves that did not adhere to the three space rule could be removed. This was achieved by finding a path from the potential move to the Spider and if no path with length three existed, then the move was discarded.

The Grasshopper's moves were more complicated than the other pieces and as such the moves had to be calculated independently. This was accomplished by creating paths over the hive following a certain direction and stopping when a blank space was found.

The Beetle and Queen Bee's moves were calculated by taking all surrounding blank spaces and removing any that would disconnect the hive. The Beetle, however, also required moves that placed it onto another piece. Furthermore, when the Beetle covered a piece then it could

move to any space surrounding it, without risk of disconnecting the hive. The event handler for movement had to account for four distinct cases for the Beetle. The first, if a Beetle was moving onto another piece. The second, if a Beetle was moving off a piece. The third, if a Beetle was moving off a piece and onto another piece and the fourth, if a Beetle was moving to an empty space and not covering a piece.

3.4 AI Implementation

An AI player required the functionality to evaluate the next set of moves and return the best move it could make. For this project, a basic AI player was designed first before adding more complex heuristics.

3.4.1 Basic AI Player

For the basic AI player a single heuristic was used. Potentially the most powerful single heuristic for the game Hive, was the number of pieces surrounding a player's Queen Bee minus the number of pieces surround the opposing player's Queen Bee. This could be the best single heuristic to use as the main objective of the game is to surround the opposing player's Queen Bee. Therefore, it was the best choice for the basic AI player.

3.4.2 Advanced AI Player

The advanced AI player utilised multiple different heuristics to form one heuristic. This was then used to evaluate the next moves available at a given state. A total of fifteen heuristics were devised. The heuristics largely consisted of the number of pieces of a given creature that could move. For example, the number of black Spiders that could move. These *movement heuristics* comprised twelve of the total heuristics, six for either player's pieces. This fits with the core of the game as the mobility of a player allows them to win more. For the heuristics that measured the opposing player's mobility, a negative multiplier was applied so moves were not made that would have benefitted the opposing player. The remaining heuristics accounted for the number of pieces surrounding either player's Queen Bee as well as the overall number of movable pieces a player had. The last heuristic calculated if the move would place a Beetle onto another piece. This was important as a covered piece cannot move. Weights were created for each heuristic to allow for certain heuristics to be more important than others. For instance, the mobility of a Soldier Ant is generally more important than the mobility of a Grasshopper, as a Soldier Ant has more movement options and therefore, can be used more strategically. When evaluating a move, the advanced AI player also checked if the move was a winning one. If this was the case, then the move's heuristic value was increased greatly. This meant that if a winning move could be made, the AI player would select it.

3.4.3 Minimax Implementation

Minimax was implemented as a function in the *HiveGame* class. If an AI player used the minimax strategy then the function was called when determining the next move. The function worked as described in Section 2.2.4. The branching factor for the game is large as the pieces have a

variety of moves that can be made. The Soldier Ant, for example, has a move for each space around the perimeter of the hive. This leads to an average branching factor of 50, although it can be much higher in certain states. Alpha-Beta Pruning was added to lower the branching factor and make computation quicker. The children generated from a node were also ordered by heuristic state evaluation so that the alpha-beta pruning would perform better. Due to the large branching factor of the game and the time taken to explore nodes, a limit was imposed to reduce the number of children explored. The top 10 children were evaluated for each node, with each child being evaluated by the AI player's heuristics.

3.4.4 Genetic Algorithm Implementation

The genetic algorithm was implemented via the *GeneticAlgo* class. Chromosomes were encoded as a list of weights corresponding to the heuristics of an AI player. The *crossover* function took two parent chromosomes and randomly selected a point to crossover. The child chromosome was the same as the first parent chromosome prior to the crossover point and the same as the second parent chromosome after the crossover point. The *mutate* function took a chromosome and randomly selected a point in the chromosome, the mutation point. It then randomly selected a value between -0.1 and 0.1. This value was the mutation amount. The mutation amount was then added to the value at the mutation point.

When creating a new generation, the parents were randomly selected with preference toward the chromosomes that won more games. The two parents were then passed into the crossover function to produce a child. It was then randomly decided whether to mutate the child or not. Every chromosome in a generation was played against each other using the *Game Simulator* component. Each time a chromosome won a game, the score for that chromosome increased. If the chromosome won within a certain number of turns, a multiplier was applied. This weighted chromosomes that win in less turns as more important and thus, they were more likely to be picked to become parents of the next generation.

3.4.5 Reinforcement Learning Algorithm Implementation

The reinforcement learning algorithm was implemented with the *ReinforcementAlgo* class. It used passive learning techniques (discussed in Section 2.2.6) to discover the optimum weights for the heuristics of an AI player. The parameters were encoded as a list of weights, similarly to the chromosomes in the Genetic Algorithm Implementation (Section 3.4.4). The algorithm learnt by playing multiple games while varying the values of the parameters. The parameter configuration that performed the best after playing all games was then selected for the next iteration. If none of the games played resulted in a win, then the values of the parameters were randomly changed. This corresponded to the agent exploring its environment. The games were played using the *Game Simulator* component.

3.4.6 Optimisation

A single game between two AI players took, on average, approximately six minutes to run. Running the reinforcement learning algorithm and the genetic algorithm required numerous

games to be executed, so a more efficient solution was required. Parallelism was implemented, through the use of sub-processes, so that multiple games could be run at the same time. This meant that five games could be executed in the same time a single game would take to run. Furthermore, the reinforcement learning and genetic algorithm would terminate in a fifth the time and thus, training an AI agent would be much faster.

Chapter 4

Evaluation

4.1 Testing Strategy

To evaluate the heuristics used a strategy had to be devised. The strategy was represented by a certain weighting of parameters for each heuristic. Multiple games were run with this configuration. In addition to this, the basic AI player was played against different configurations of the advanced AI player. To ensure games would not run indefinitely, a turn limit of 150 was imposed. If a game reached 150 turns, the game was terminated and the outcome recorded as a draw.

To test the minimax implementation, multiple games were run with a set depth of one and using a set strategy to determine the effectiveness of the minimax implementation. The main strategy used to test the different AI learning techniques was to run the algorithms and compare the outcomes to the initial solutions. The results of this comparison were then used to ascertain whether the algorithms had learnt a better solution. The best outcomes from each technique were played against each other. This meant that the most successful chromosome produced from running the genetic algorithm was played against the most successful parameter configuration produced from the reinforcement learning algorithm. Both of the best outcomes from each technique were also played against the manually created strategies. The result of these games showed the better configuration of weights. As discussed in Section 4.4 and Section 4.5, both learning algorithms did not produce successful solutions so certain tests were not carried out. To ensure the resulting solution was effective it was intended to test the successful solution against human players. Due to the COVID-19 outbreak, this was not possible (further discussed in Appendix B).

4.2 Heuristic Evaluation

The first test run was two advanced AI players with neutral weights playing against each other, intended as a baseline. This meant that all further tests could refer back to it to measure the relative effectiveness of an advanced AI player. Out of 100 games, 83 were terminated for exceeding the turn limit. The number of turns taken to win were inconsistent, with a range of 16 to 62. The white player won eight games and the black player won nine games. This shows that, with no inherent strategy or goal, a neutral set of weights will still win a small number of games.

A test, comparable to the previous, was run, but with two basic AI players. This was carried out to test the effectiveness of a basic AI player. There were 59 wins out of 100 games, with the white player winning 34 games and the black player winning 25 games. This further supports the outcome of the previous test. The basic AI players used the simplest strategy of selecting moves that would surround the opposing Queen Bee. The advanced AI players require

a strategy to improve their effectiveness. A test was run playing the advanced AI player, with neutral weights, against the basic AI player, using a single heuristic. The results are shown in Figure 4.1a. The advanced AI player won six times and the basic AI player won 24 times. This continues to demonstrate the notion that the advanced AI player requires a strategy to be effective.

A further test was run with the advanced AI player, using a manually determined set of weights. This configuration weighted the heuristics that determine the number of pieces surrounding each player's Queen Bee twice as important than the other heuristics. Results are shown in Figure 4.1b. The advanced AI player won 54 games and the basic AI player won 17, with the remaining 29 games resulting in a forced termination. The advanced AI player won more than double that of the basic AI player. This indicated that, with an appropriate strategy, represented by the set of weights, the advanced AI player performs much better than the basic AI player. The strategy, in this situation, was a balance between focusing on surrounding the other player's Queen Bee while protecting one's own.

4.3 Minimax Evaluation

Due to the children exploration limit enforced on the algorithm, discussed in Section 3.4.3, only the top 10 children for each node were explored. This meant that the algorithm relied heavily on the heuristic evaluation of children to be accurate. Therefore, results shown by the minimax test may not be reliable nor representative of the algorithm's potential to improve an AI player's performance. The test played one advanced AI player, using the manually determined set of weights with minimax and a depth of one, against another advanced AI player, with neutral weights. The results can be seen in Figure 4.2a. Only 11 games were won out of 100, with the minimax player wining six. When compared to the previous tests run in Section 4.2, the minimax player performed poorly. The advanced AI player with manually determined weights won 54 games in previous tests, so the minimax algorithm must have made the player less effective. This could be due to the limitations of the implementation, the depth limit, or a combination of both factors. The heuristic performed well when not used in the minimax algorithm which suggests that the depth limit may be the prevailing cause of the poor performance.

4.4 Genetic Algorithm Evaluation

The genetic algorithm was first run with a population size of 21 for six generations. The results can be seen in Figure 4.3a. Initial observations may lead to the conclusion that chromosome four in generation five was most successful as it had the highest score of 25. However, the scores achieved do not directly correlate with number of games won as the scores were calculated with a multiplier. This multiplier value increased inversely with the number of moves taken to win a game. Therefore, higher scores do not necessarily equate to a better configuration of parameters. The chromosome that was most successful was the one that achieved high scores consistently and accurately. Following this criteria, the most successful chromosome overall would be chromosome 16.

Chromosome 16 was then evaluated by playing against an advanced AI player, with a neutral set of weights, for 100 games. The results are shown in 4.3b. Notably, it can be seen that the majority of games resulted in a forced termination, suggesting that the chromosome was not particularly effective. However, it won in a consistent number of turns, with few anomalies. This could be explained by likening the result to strictly executing the same strategy each game. The strategy may only be effective in particular scenarios and unsuccessful in most others, so the chromosome has likely learnt a specific strategy.

An issue with genetic algorithms is that, when applied to complex problems, they can take numerous generations before they can learn a general solution. This is because the algorithm initially contains a substantial amount of randomness. If the initial generation does not contain a good solution then the algorithm will need to continuously breed more generations before a good solution presents itself. For this reason, the genetic algorithm was executed again with a smaller population size of 11 for 15 generations.

The results of the second run of the genetic algorithm can be seen in Figure 4.3c. They show that the generations, on average, improve. The chromosomes in generation 15 scored higher, with a mean score of 2.55, than the chromosomes in the first generation, which had a mean score of 2.27. A statistical t-Test was run to determine whether the difference was significant. However, the difference was non-significant, $t(20)=0.24$, $p=0.41$. It is probable that running the algorithm for more generations would result in a substantial increase in scores achieved. This supports the above point, as the second run of the algorithm improved when more generations were used.

4.5 Reinforcement Learning Algorithm Evaluation

The reinforcement learning algorithm was first run for five iterations, with 15 different configurations. There was one configuration for each variation on each parameter. Results can be seen in Figure 4.4a. The results show little data and could suggest five iterations were too low for the algorithm to converge. As a result of this, the algorithm was run again for 60 iterations. At approximately 47 iterations, the algorithm learnt the opposite of its goal. It started to use configurations that helped the opposing player to win. The cause of this deviation was most likely due to the exploration component in the algorithm. When the algorithm does not make progress for an iteration it randomly varies the parameters, resulting in a configuration of parameters that was detrimental to the learning process. This meant that, from iteration 47, the algorithm was learning from a negative position, effectively starting the learning process again. The effect of this caused the consecutive iterations to make little progress.

A configuration was picked from iteration 47, further referred to as I47-Alpha, and played against a neutral set of weights for 100 games. The results are shown in Figure 4.4b. The neutral set of weights won 53 out of 100 games, on average winning in 37 moves. When evaluating the heuristics (Section 4.2), it was shown that playing two sets of neutral weights against each other resulted in 83 forced terminations. This shows that I47-Alpha was a contributing factor towards

the winning games.

As I47-Alpha achieved the opposite of a rational player, a test was run using the inverse of I47-Alpha, further referred to as I47-Beta, to determine if the inverse could play rationally. I47-Beta used weights that were the multiplicative inverse to the weights in I47-Alpha. The results are shown in Figure 4.4c. The neutral set of weights won 97 games, with an average number of moves of 34. I47-Beta won no games. I47-Beta is a clear improvement on I47-Alpha, if the goal was to help the opposing player to win. However, knowing the set of weights which results in the opposing player winning could be useful. It could provide insight into a set of weights that will allow the player to win with the same rate and average number of moves. Despite this, it could not be significant in any way.

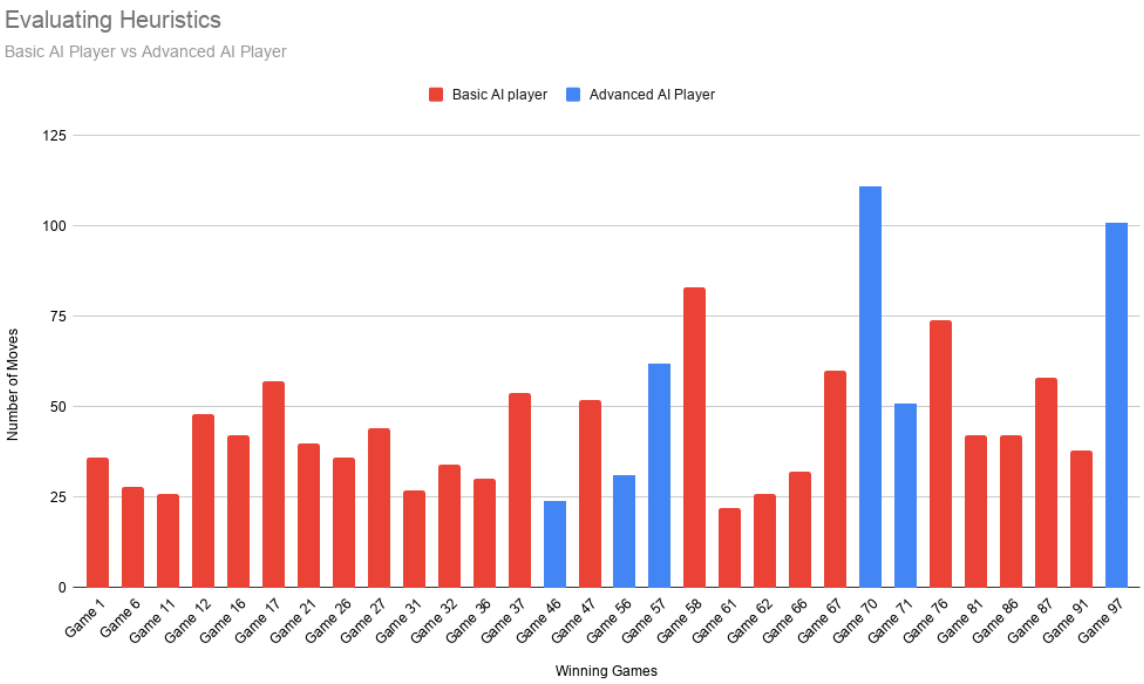
4.6 Comparison of AI Techniques

4.6.1 Basic AI Player and Advanced AI Player

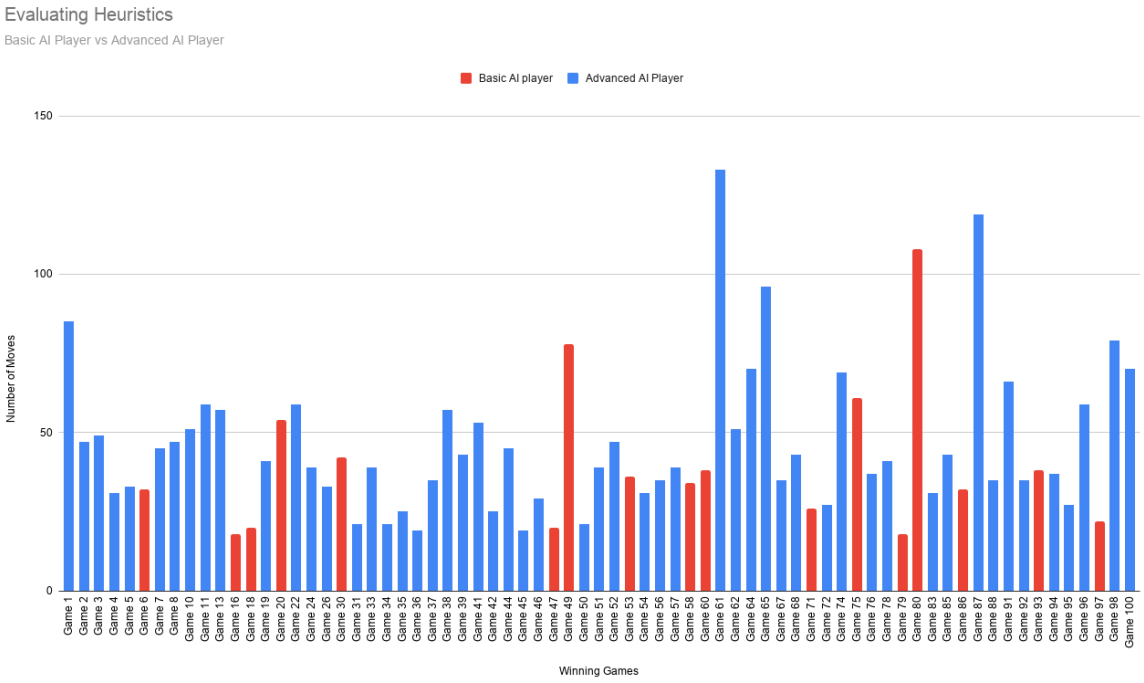
From the tests run in Section 4.2 it can be observed that the advanced AI player performs better than the basic AI player, when given a valid strategy. The additional information available to the advanced AI player allows it to make better decisions so it is more likely to win, as shown in Figure 4.1b. The basic AI player performs best when the opposing player does not use a clear strategy. This is shown by the tests that played the basic AI player against an advanced AI player with neutral weights. This indicates that the use of multiple heuristics to inform decisions made by an AI agent is only useful if the heuristics are chosen and weighted appropriately. In absence of multiple heuristics, a single heuristic can be used, if that heuristic accurately represents the main goal of the game. For example, the basic AI player used a heuristic that represented the number of pieces surrounding each player's Queen Bee. The goal of the game Hive, is to surround the opposing player's Queen Bee and as such, this heuristic is appropriate to be used without additional information.

4.6.2 Genetic and Reinforcement Learning Algorithms

Both algorithms failed to converge to a suitable solution. However, the tests run in Section 4.4 show that the results of the genetic algorithm were better than the initial solutions. The tests run in Section 4.5 show that the algorithm's final solutions did not improve on the initial solution. Furthermore, during the learning process the solutions got worse. This suggests that the genetic algorithm performed better when applied to the game Hive. This could be because the main elements of randomness in the genetic algorithm existed at the beginning, when the initial generation was created, and each subsequent generation was created from the best solutions in the previous. The mutation operation added another element of randomness to the genetic algorithm, but the change was not enough to impact the solutions negatively. The reinforcement learning algorithm's solution could be detrimentally affected by the exploration aspect of the method. This changes the entire solution randomly and impacts all future solutions, hence why the exploration that resulted in I47-Alpha caused the following iterations to perform poorly.

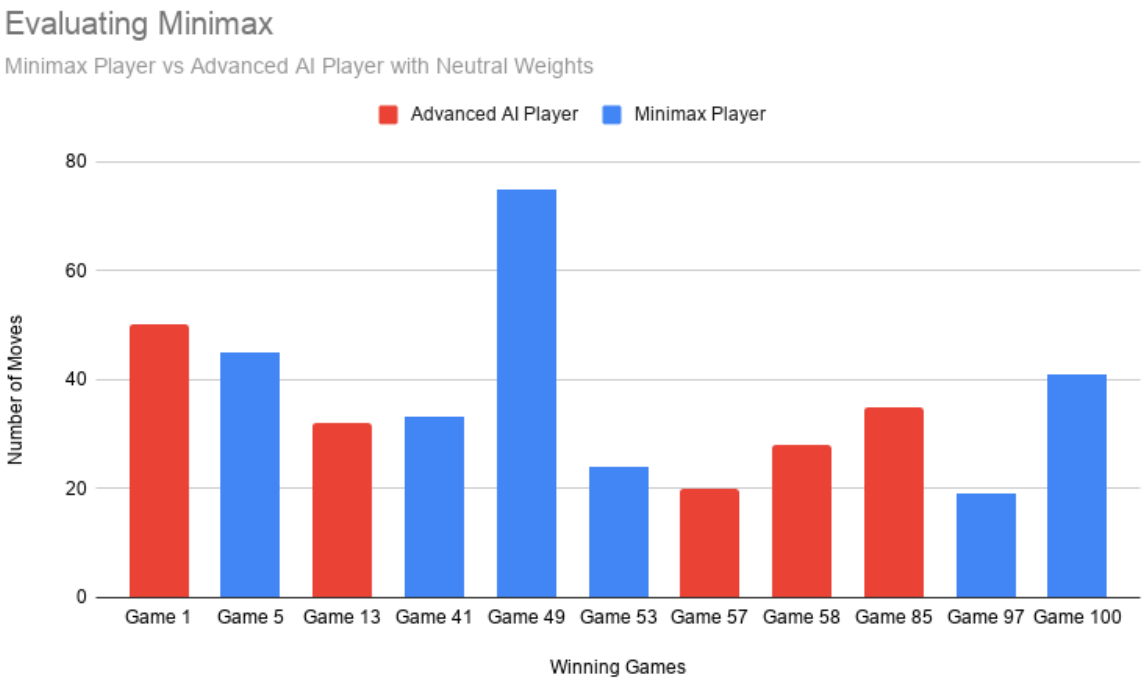


(a) Neutral Set of Weights



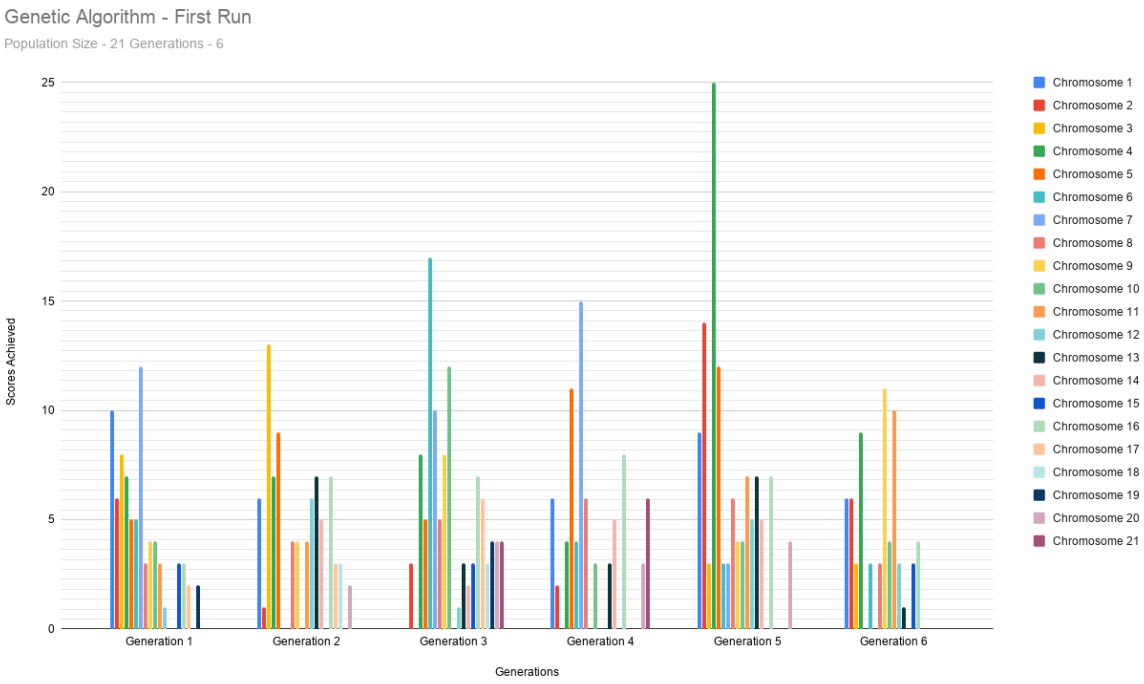
(b) Manually Determined Set of Weights

Figure 4.1: Advanced AI Player Against Basic AI Player

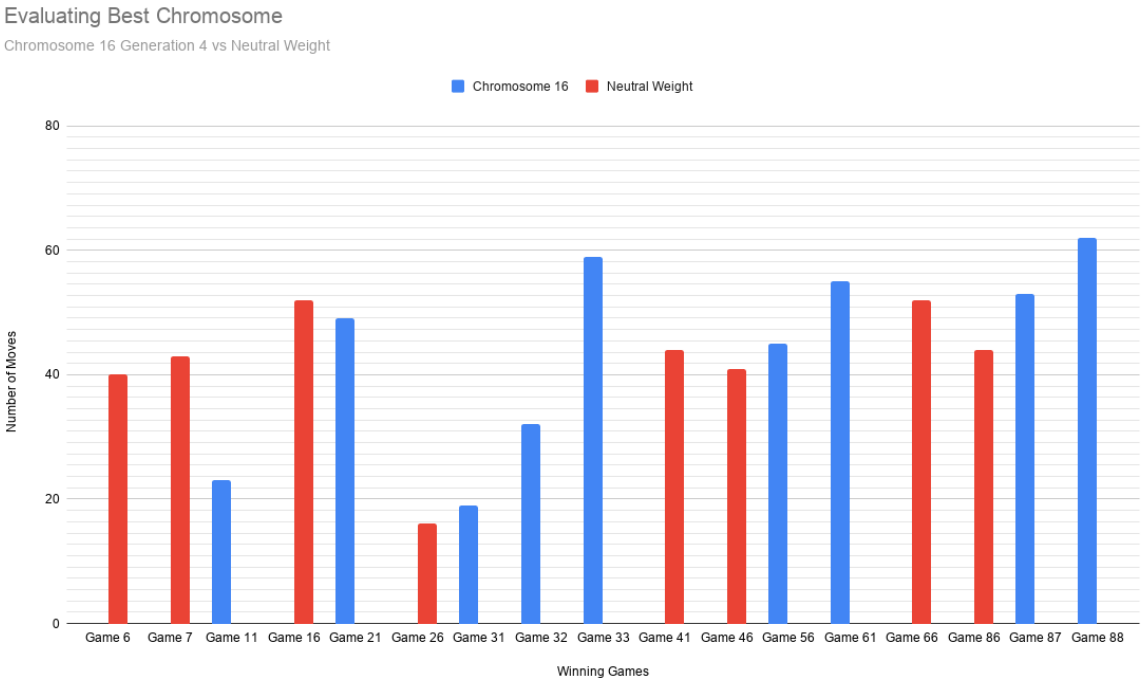


(a) Minimax Player Test Results

Figure 4.2: Minimax Results

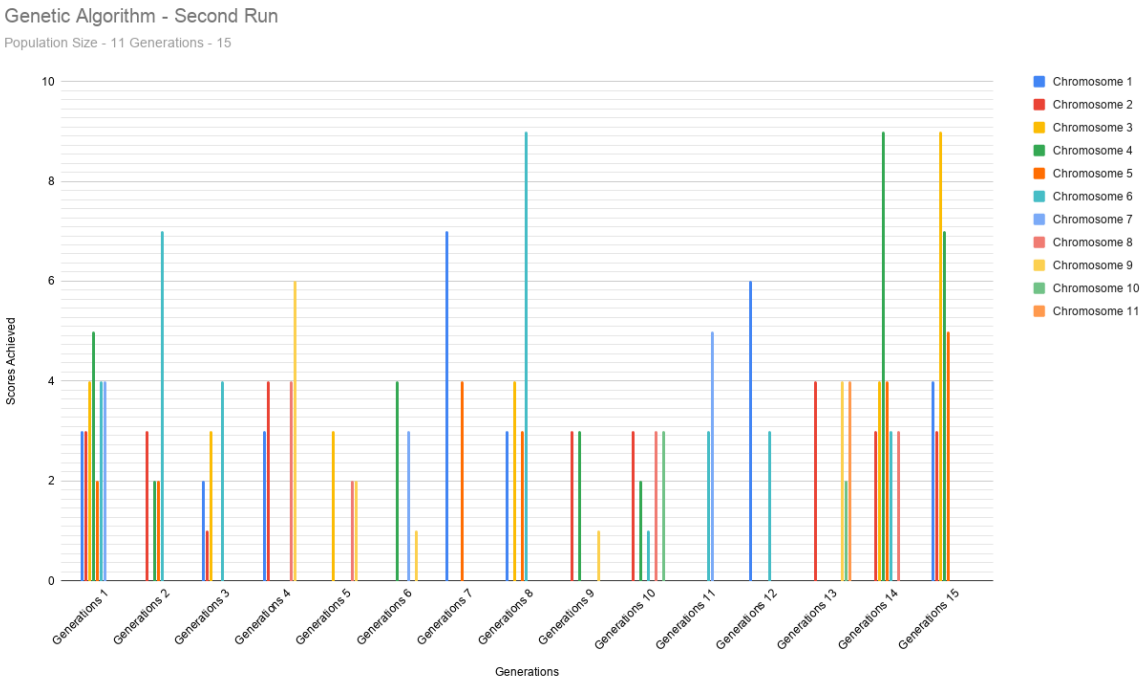


(a) Genetic Algorithm - First Run



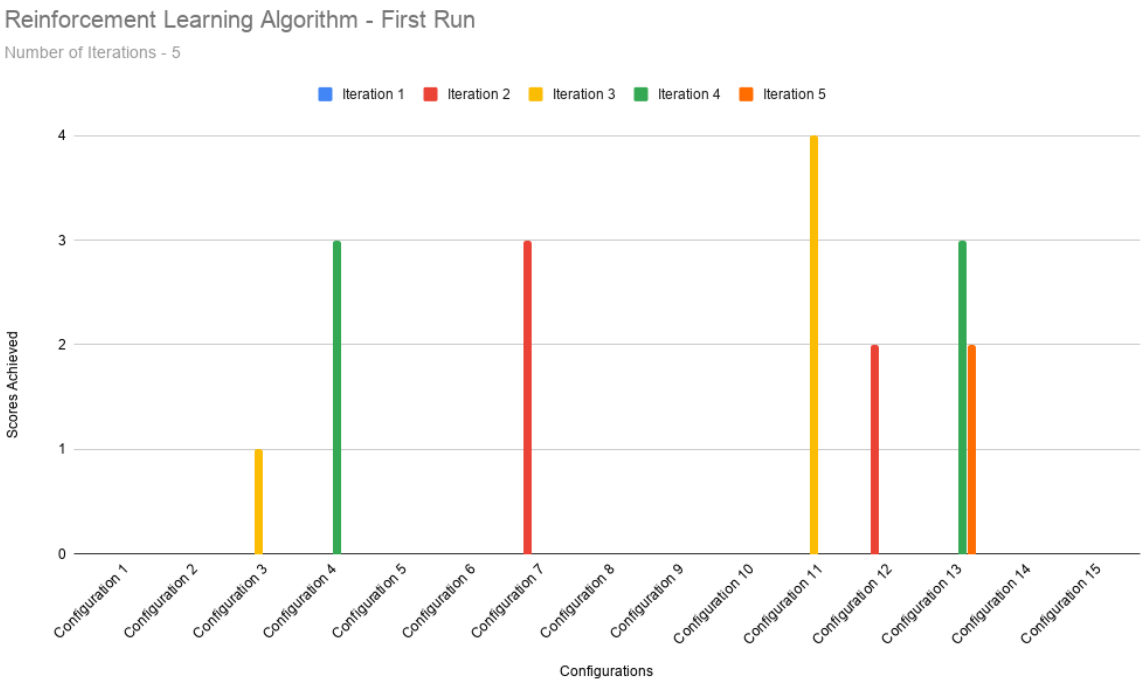
(b) Best Chromosome Test Results

Figure 4.3: Genetic Algorithm Results

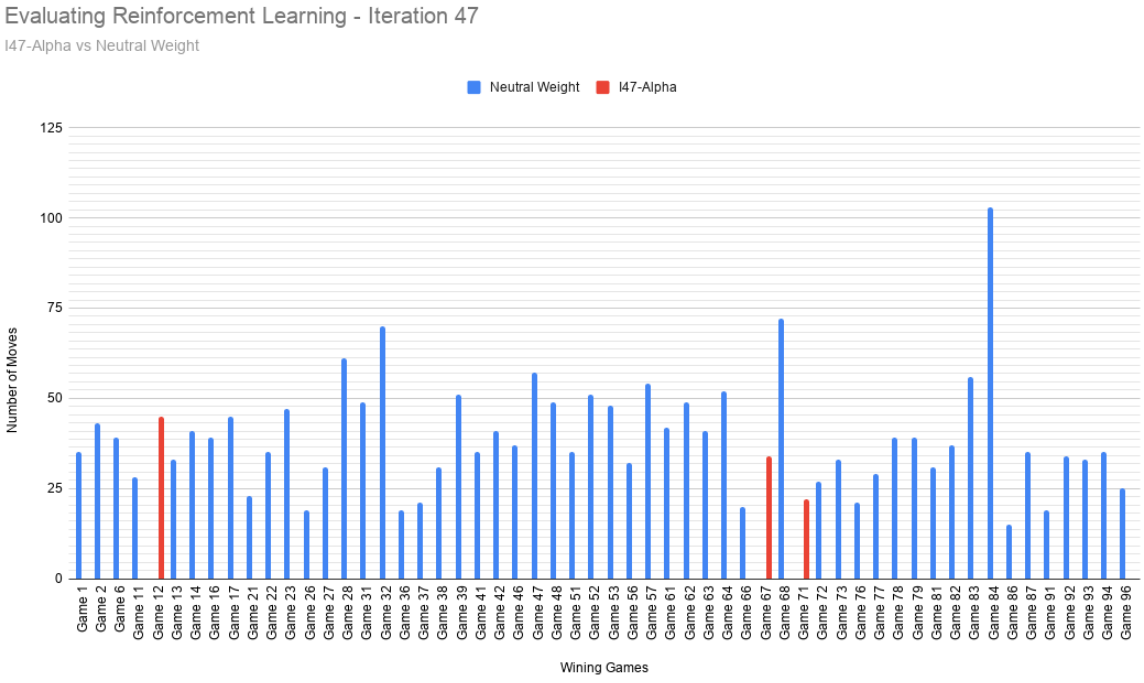


(c) Genetic Algorithm - Second Run

Figure 4.3: Genetic Algorithm Results

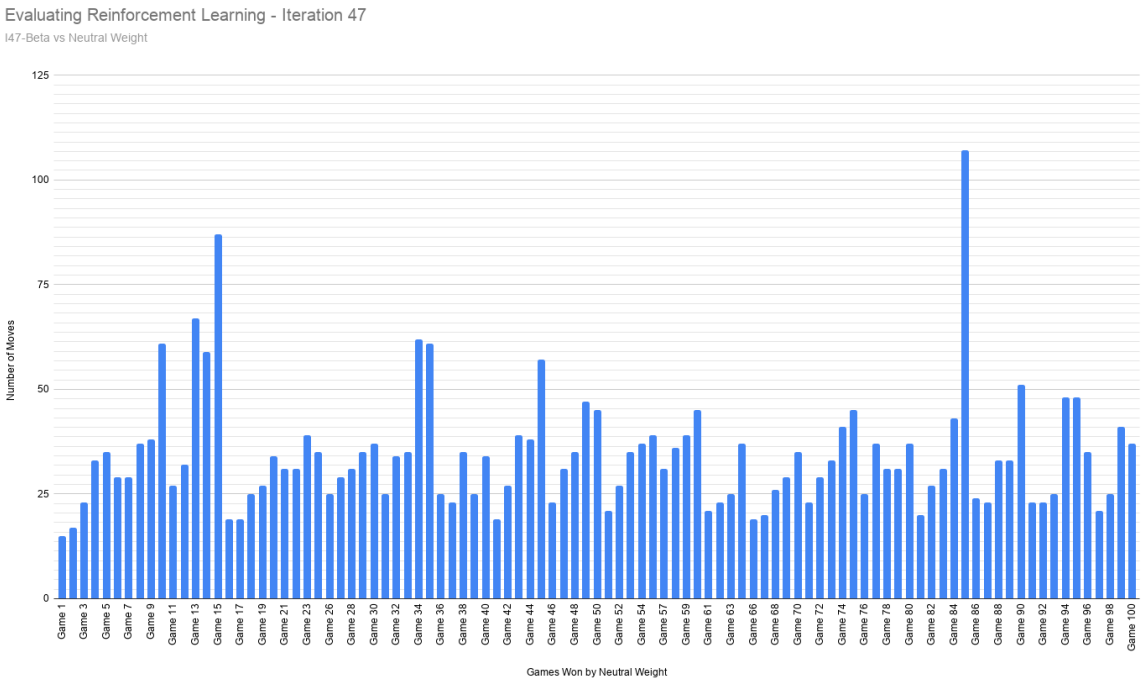


(a) Reinforcement Learning Algorithm - First Run



(b) I47-Alpha Test Results

Figure 4.4: Reinforcement Learning Algorithm Results



(c) I47-Beta Test Results

Figure 4.4: Genetic Algorithm Results

Chapter 5

Legal, Ethics, Social and Professional Issues

This project has been developed in accordance with the British Computer Society (BCS) code of conduct [38].

5.1 Ethical Issues

The development of the board game Hive does not have any associated ethical issues. However, the implementation of AI techniques to play the game requires ethical consideration. As AI agents become increasingly popular and exponentially more successful, the ethical issues that must be reviewed become more numerous. One such issue concerns decisions of automated vehicles during crashes. Humans, when making decisions during a crash, will aim to minimise the damage that can occur to them. These decisions are usually poor due to the conditions disallowing appropriate planning. Contrasting with this, an automated vehicle will be able to anticipate the oncoming crash and make an informed decision. This decision could have dire consequences for the driver or any nearby vehicles. In situations where a crash cannot be avoided, the system would have to decide the course of action that will cause the least damage. This immediately raises ethical issues such as the value of life and which life is more important in saving [12].

Another issue relates to game playing AI agents and the responsibility of their creators to instil them with ethics [44]. Consider a reinforcement learning agent that maximises reward. This agent is used in a first-person shooting game with the goal of killing the most enemy combatants to receive a reward. Studies have shown that an AI agent can be trained to outperform human players in both single player and multiplayer scenarios [19]. If this scenario is abstracted to a military excursion the problems become apparent. Programmers have the ability to create single-minded agents that never stop maximising a reward which can result in severe damage to, and potentially loss of life. It is the obligation of any AI developer to ensure that the resulting AI will include a strict code of ethics at its core or the outcomes could be disastrous.

5.2 Legal Issues

The only legal issue that could be considered is the one of intellectual property (IP) and copyright for the board game Hive. Gen Four Two Ltd owns the copyright for the game and it is their IP. This would be an issue if I were to distribute or aim to profit from my implementation of the game. However, my use of the game is purely academic and for non-commercial use and as such, does not infringe on their copyright. This also acts in accordance with the BCS code of conduct regarding the "Duty to Relevant Authority" and "Professional Competence and Integrity" sections [38].

5.3 Social Issues

There are no social issues to consider with this project. Social issues arise when systems discriminate against people based on someone's characteristics. In this project the only opportunity for discrimination presents itself from the AI agents playing the game. The AI agents are given no information about the entity they are playing against and therefore, cannot differentiate between people and act differently according to whomever they are playing against.

5.4 Professional Issues

The BCS code of conduct states that work should only be undertaken when the worker has the professional skills to carry out the work accurately and competently [38]. This has been adhered to as the initial objectives and aims of the project have been successfully met, showing that the project was achievable with the professional skills I possessed. The project has been developed to a professional standard, shown by the design, implementation, testing and evaluation of the solution to fulfil the aims and objectives. Consequently, there are no professional issues regarding this project.

Chapter 6

Conclusion

6.1 Aims and Objectives

The primary aim of this project, described in Section 1.2 was met when the Hive AI component was implemented and integrated into the Hive Game component. The secondary aim was met when developing the Game Simulator, Genetic Algorithm, and Reinforcement Learning Algorithm components.

Four main objectives were established, outlined in Section 1.3. The first, to develop software allowing users to play the game Hive. This was met when the Hive Game component was developed, as described in Section 3.3. The second objective was to implement an algorithm that could simulate a player, which was met when the functionality allowing for generation and selection of moves was implemented. A sub-objective was to develop software that allowed for two AI agents to play against each other. Although separate software was not developed, the functionality was implemented through the Hive AI and Hive Game components, so the objective was met. The third objective was to use AI techniques to improve the algorithm and to create more algorithms. This was met when developing the Minimax strategy, Genetic Algorithm, and Reinforcement Learning Algorithm components, explained in Section 3.4. The final objective was to evaluate the algorithms and compare them to each other. This was met when testing the system and analysing the results, shown in Chapter 4.

6.2 Further Development

The main problem this project faced was the time required to run a game with two AI players. This, in turn, resulted in the incredibly long training times for AI players. The first task for any further development would be to modularise the project further. This would ensure that, when training AI players, there would be no computation wasted on the front end graphics of the game. Additional tasks would include further testing of the AI techniques and improving the minimax implementation. The test results in Section 4.4 and Section 4.5 suggested that the learning algorithms required more testing to produce more substantial, significant results. The minimax implementation could be improved by refactoring the code and removing the limits enforced, discussed in Section 3.4.3, to achieve faster runtimes and potentially, a more powerful algorithm.

6.3 Personal Reflection

Overall, I believe this project has been carried out competently, with all aims and objectives met. The fields of AI and games are ones that I find very intriguing. This helped for research and planning the project as I had genuine interest in learning. The project has been stressful

and enjoyable and I have learnt a lot and gained a lot of experience.

The main issues I faced during this project were timing, implementing complicated movement options for the game and developing the AI player. I also underestimated the time required to write the final report. The problems with implementing some of the movement options with the game stemmed from difficulty with integrating the logical framework of the game and the graphical framework provided by TkInter. The task of creating an AI player was quite overwhelming as it was something I had never attempted before. However, I was also excited by the prospect of creating an intelligent agent as I have a great appreciation of AI.

In future projects, to avoid these problems I will allow for more time to write the final report as well as write the report more incrementally. I will ensure that I fully understand all of the technologies I am working with and create prototypes to test features before implementing them in the final system.

References

[1] Hive game rules. https://www.ultraboardgames.com/hive/game-rules.php. Accessed: 23-09-2019.

[2] E. Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[3] R. Bellman. *An introduction to artificial intelligence: Can computers think?* Thomson Course Technology, 1978.

[4] H. J. Berliner. Some necessary conditions for a master chess program. In *IJCAI*, volume 3, page 77, 1973.

[5] P. Borovska and M. Lazarova. Efficiency of parallel minimax algorithm for game tree search. In *Proceedings of the 2007 international conference on Computer systems and technologies*, pages 1–6, 2007.

[6] S. J. Brams. Theory of moves. *American Scientist*, 81(6):562–570, 1993.

[7] I. Brocas, J. D. Carrillo, and A. Sachdeva. The path to equilibrium in sequential and simultaneous games: A mousetracking study. *Journal of Economic Theory*, 178:246–274, 2018.

[8] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[9] A. M. Colman. *Game theory and experimental games: The study of strategic interaction*. Elsevier, 2016.

[10] M. Gen and L. Lin. Genetic algorithms. *Wiley Encyclopedia of Computer Science and Engineering*, pages 1–15, 2007.

[11] Gen42Games. Hive. https://www.gen42.com/games/hive. Accessed: 23-09-2019.

[12] N. J. Goodall. Ethical decision making during automated vehicle crashes. *Transportation Research Record*, 2424(1):58–65, 2014.

[13] J. H. Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[14] M. O. Jackson. A brief introduction to the basics of game theory. *Available at SSRN 1968579*, 2011.

[15] R. E. Korf. Heuristic evaluation functions in artificial intelligence search algorithms. *Minds and Machines*, 5(4):489–498, 1995.

[16] R. E. Korf. *Artificial intelligence search algorithms*. Computer Science Department, University of California, 1996.

[17] H. W. Kuhn and A. W. Tucker. *Contributions to the Theory of Games*, volume 2. Princeton University Press, 1953.

[18] R. Kurzweil, R. Richter, R. Kurzweil, and M. L. Schneider. *The age of intelligent machines*, volume 579. MIT press Cambridge, 1990.

[19] G. Lample and D. S. Chaplot. Playing fps games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[20] K. Leyton-Brown and Y. Shoham. Essentials of game theory: A concise multidisciplinary introduction. *Synthesis lectures on artificial intelligence and machine learning*, 2(1):1–88, 2008.

[21] S. Marsland. *Machine learning: an algorithmic perspective*. CRC press, 2015.

[22] M. Mitchell. L.d. davis, handbook of genetic algorithms. *Artificial Intelligence*, 100(1):325–330, 1998.

[23] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[24] R. B. Myerson. *Game theory*. Harvard university press, 2013.

[25] J. Nash. Two-person cooperative games. *Econometrica: Journal of the Econometric Society*, pages 128–140, 1953.

[26] M. Newborn. *Beyond deep blue: Chess in the stratosphere*. Springer Science & Business Media, 2011.

[27] M. J. Osborne et al. *An introduction to game theory*, volume 3. Oxford university press New York, 2004.

[28] M. J. Osborne and A. Rubinstein. *A course in game theory*. MIT press, 1994.

[29] T. Roughgarden. Algorithmic game theory. *Communications of the ACM*, 53(7):78–86, 2010.

[30] S. Russell and P. Norvig. *Artificial intelligence: A Modern Approach*. Pearson Education Limited, 2002.

[31] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[32] A. P. Saygin, I. Cicekli, and V. Akman. Turing test: 50 years later. *Minds and machines*, 10(4):463–518, 2000.

[33] C. Shapiro. The theory of business strategy. *The Rand journal of economics*, 20(1):125–137, 1989.

[34] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[35] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[36] S. Sivanandam and S. Deepa. Genetic algorithms. In *Introduction to genetic algorithms*, pages 15–37. Springer, 2008.

[37] J. M. Smith. The theory of games and the evolution of animal conflicts. *Journal of theoretical biology*, 47(1):209–221, 1974.

[38] B. C. Society. British computer society code of conduct. *Professional Issues in Software Engineering*, pages 199–205, 1991.

[39] I. Sommerville. *Software Engineering Ninth Edition*. Pearson, 2009.

[40] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[41] A. M. Turing. Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer, 2009.

[42] J. Von Neumann, O. Morgenstern, and H. W. Kuhn. *Theory of games and economic behavior (commemorative edition)*. Princeton university press, 2007.

[43] P. Winston. *Artificial Intelligence (Third edition)*. Addison-Wesley, 1992.

[44] H. Yu, Z. Shen, C. Miao, C. Leung, V. R. Lesser, and Q. Yang. Building ethics into artificial intelligence. *arXiv preprint arXiv:1812.02953*, 2018.

Appendices

Appendix A

External Material

A.1 Code Repository

The code developed for this project is accessible on *GitLab* from the URL:
`https://gitlab.com/ll16ain/final-year-project`

A.2 Build Instructions

The instructions to build and run the project are available at:
`https://gitlab.com/ll16ain/final-year-project/-/wikis/Build-instructions`

Appendix B

Impact Report

Due to the global pandemic, caused by COVID-19, human participants could not be used for testing of the software and different AI techniques. This resulted in reduced testing for the project which could be pursued in further study (Section 6.2).