School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Final Report

AI for Board Games

Marya Amina Khan

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

2018/19

40 credits

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Dissertation	*Report x 2*	*SSO (26/04/19)*
Monopoly Code	*Software codes*	*Supervisor, assessor (26/04/19)*
Monopoly Code	*Git URL*	*Supervisor, assessor (26/04/19)*

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 investigates various *artificial intelligence* techniques as well as an *artificial intelligence* player that can play the board game *Monopoly*.

Acknowledgements

I would like to thank God as this project would not have been completed without his guidance and love.

Thank you to Dr Brandon Bennett for his invaluable support throughout this project.

Finally, my Mother, Laurie and Aleena Siddique for their continuous support and reassurance.

Table of Contents

Chapter One

Project Outline

1.1 Introduction

There has been much time devoted to developing Artificial Intelligence (AI) programs that achieve the same playing ability, if not better, of a human in a number of classic board games such as *Chess, Checkers* and *Connect-Four.* Subsequently, it generated much success, to the degree that many of these classic games now have programs that can beat the best human players [1][2][3]. A lot of this development has been because of their use of techniques such as *Minimax* and *Alpha-beta pruning* which had originated from *Artificial Intelligence* algorithms for game playing. This fascination with strategic games and the ability to simulate a player that chooses the most effective move (in relation to the overall outcome of the game) has captured the focus of many professionals since the start of *artificial intelligence* [4].

However, not all games have had the same success due to the fact that there are games where move-making becomes non-deterministic. This is due to the fact that the entire game sate is not visible to each player, as each player's own information is not revealed to the rest [5]. Games that fall under this category are *Poker* and *Bridge,* and are labelled a different classification of game as a consequence.

1.2 Aims

The aim of this project is to attempt to successfully create an artificial player, which possesses the optimal strategy in its tested set, to play *Monopoly.* Results from testing can be slightly inconsistent due to the unpredictable nature of the dice. When developing this project, it is expected that the most intriguing and challenging part will be trying to simulate the aspect of dealing and buying properties, all in accordance to a strategy.

1.3 Objectives

The objectives set has been used as a check-list to ensure that the project remains on track in addition to bringing organisation and structure throughout.

1. Investigate the different uses of AI techniques such as *MiniMax* and *Alpha-beta pruning*.
2. Develop a software that can play *Monopoly* in accordance to its rules
3. Develop an AI algorithm to play the game of *Monopoly*
4. Investigate which strategies are 'better' at playing the game

Figure 1.1: Monopoly Game Board

1.4 Plans

There are two types of plans that will be discussed, the long term and the short term plan. The project is structured so that much of the work is to be done in Semester 2 as there will be less workload involving assignments for other modules as well as exam preparation

Semester 1 - the focus was to shape the project conceptually by gaining knowledge on how the game would be structured. This was done through much background research with regards to existing literature of a similar nature. The decision on the choice of game to be implemented was based on several factors; the difficulty of representing the game state and which areas would be most challenging. Through looking at previous projects that were similar it was decided that *Monopoly* would provide the complexity and challenge that was wanted from this project, whilst pulling together AI techniques.

Semester 2 – the majority of the project would develop in Semester 2 as key concepts and ideas intertwine as well as most of the coding. In terms of the report write-up it has been organised so that a large section of it would be done during the Easter break.

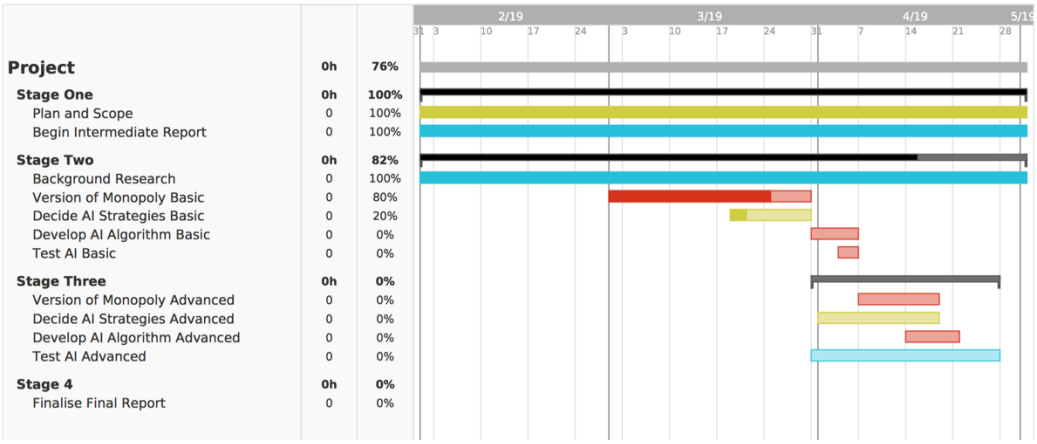

Figure 1.2: Gantt Chart for Project

1.5 Deliverable

A *Final Project Report* that will provide a literature review on *Game Theory* and AI techniques, as well as the actual design and creation process of any resultant software that will have been created. Including the details of testing results. The project will also include prototype software which implements an AI with the ability to play Monopoly as well as the subsequent results of each tested version of said software.

1.6 Minimum Requirements

Develop a prototype software that determines which AI technique can be considered most effective in playing *Monopoly*.

1.7 Methodology

Given the nature of the project, a strict regime would not be appropriate. Methods and techniques are continuously changing as more background research is accumulated. Thus, there has been little meticulous planning and a rigid structure put in place at the very beginning of this project. As traditional techniques are set out to build upon previous work, a rigid structure is not found to be the best method in this particular case [6]. A preferred agile approach was adopted because it realises that change needs to be managed, not avoided, something in which is inevitable in this project [6]. Much of the software development process will occur in Semester 2, but it will be set out in a very scrum-like method. The details of which will be decided nearer to the time. There will be changes to the software constantly and so it would only be appropriate and for testing to occur during each sprint which will allow the project to evolve naturally and smoothly.

Chapter Two

Background Research

2.1 Introduction

The need for background research is essential in order to gather the information and knowledge required to achieve the aims of this project. There has already been research and projects that have been carried out that are related to my project which can be used to aid, guide and motivate its direction. Consequently, the background research would provide a strong foundation in which my project would then build upon.

2.2 Game Theory

Game Theory has its historical origins in 1928, when Jon Von Neumann, through his analysis of the parlour games, realised the practicability of his methods when analysing economic problems. It is suggested that game theory is the study of strategic interaction between a set of individuals [7]. Strategic interaction is the principle that individuals will act accordingly to their surroundings and it is this that will help construct their strategy [8]. It considers the investigation of two or more conflicting situations including the interaction between players and the consequences of each players' actions [9].

2.2.1 Game Components

The term "game" is not only relevant in game theory but refers to a situation in which individuals or independent actors share formal rules and consequences [10]. To be able to investigate games and gaming strategies, it is vital to know the fundamental basics needed to create a game. These different components are [18]:

Rules: They provide each player with strict boundaries that must be followed as it identifies what is legal and illegal in a game. These rules help players to form strategies and discover possible moves that they can make.

Outcomes: Each game can have various possible outcomes, whereby the outcome is determined by the collective decisions made by the player/s.

Payoffs: Each outcome creates a payoff for each player, they can differ from each other. Each player wants a better payoff, the payoff that each player wants is to win the game.

Uncertainty of the Outcome: As the outcome is determined by a combination of a player's moves it creates an element of uncertainty as you are not aware of each player's move until after they have made it, leaving a sense of unpredictability particularly if the game state is either fully or partially hidden. Also, in a single player game there will be an element of chance.□

Decision-making: In order for a game to progress, players need to make decisions at each stage. These decisions then can then be analysed using game theory.

No cheating: Game theory always follow the rules and if a player is not abiding by the rules of the game it is classified as cheating.

2.2.2 Game Terminology

Game: Illustrated by a collection of rules

State: Given point in time or stage of a game using the existing components

Play: Instance of a game

Move: A decision made by the player at a given state

Strategy: Is a complete collection of moves for each state [11]. This plan influences a player's decision making.

Rational Behaviour: Each player will try to increase their chances of a better payoff whilst being aware that their opponent is attempting to do the same.

2.2.3 Game Classifications

Perfect and Imperfect Information Games

A game has *perfect information* if every player has full knowledge and is perfectly informed of all previous events leading up to the current game state, for all its players, and are able to select their next move using the information provided. *Perfect information* games provide the same level of information to each of their players [12]. *Monopoly* is a *perfect information* game as each player's property cards are on full display, their cash is fully visible to all players and each game state is on full show. This visibility influences a player's next move and strategy as they have full knowledge of the cards or pieces that each player holds and the payoffs for each possible move. For example, in *Monopoly* each player would decide to accept a deal based upon whether the payoff would be good for them. More experienced players would also consider their opponents' payoff too.

In contrast, *imperfect information* games are games where each player has little or no knowledge about any state. Players are unaware of the actions or moves made by their opponents but they can predict their opponents' strategy or future actions [13].

It is vital to know the difference between *perfect* and *complete information* games. In *complete*

information games, all players will be aware of the structure of the game but may not be able to see all of the moves made previously by the other players [14]. For example, in a card game every player is aware of the objectives of the game but each player keeps their cards hidden, this would be classed as an *imperfect* but *complete information* game.

Deterministic and Stochastic Games

A game is considered *deterministic* if whenever a player performs a move in a specific state in a game, the consequence of that move will always remain the same. Any moves made by any of the players are not subject to chance [15]. There are no other influences or elements of randomness that determines the outcome of the move. Subsequently, *Monopoly* is not *deterministic* due to the randomness of a dice, there is an element of chance and luck [15]. *Deterministic* games are *Chess, Checkers* and *Go* etc. If, for example, we take the game *Chess*, the possible outcomes of a move purely depend on the position of the other pieces on the board and the successive states that lead up to the current state [16]. The list of possibilities only depends on those two things, there is no element of randomness.

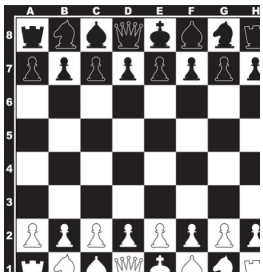

Figure 2.1: Chess Game Board

On the other hand, games that do possess that element of luck and randomness would be labeled as *non-deterministic* or *stochastic*. This does include games where none of the players are making a choice but it is influenced by an outside factor such as a roll of a dice or shuffling of cards. For example, the game *Backgammon,* which is a two player, *zero sum* game*,* is also *stochastic* because before each player makes their next move it depends on a roll of a dice. Therefore, the set of possible moves for the player is solely determined upon the randomness and luck of the dice [17]. We could define the notion of randomness as providing little predictability on the outcome of events or the lack of pattern. However, it is key to note that there is research that suggests that a dice is not completely random and that games can contain both *deterministic* and *non-deterministic* elements.

Zero-Sum or Non-Zero Sum

A game is called *zero-sum* if the sum of payoffs equals zero for any outcome [18]. Essentially, this means that whenever a player gains it is equivalent to the opponent's loss, so the net difference is zero [19]. Therefore, in order to get a positive payoff another player needs to have a negative one [18]. A

well-known example of a *zero-sum* game would be *Matching Pennies*, where you have a payoff of 1 for player A if the pennies match and so a -1 payoff for Player B and vice versa, leaving the net payoff will always be zero [18]. It should be pointed out that *Matching Pennies* is also a *symmetric game*. *Zero-Sum* games are less common than its counterpart, *non-zero sum*.

		Player B	
		Heads	Tails
Player A	Heads	1	-1
	Tails	-1	1

Figure 2.2: Payoff in Matching Pennies

Non-Zero Sum games represent the dynamics of real world problems, as sometimes no optimal solution can be found and things are not as straight forward [20]. Having multiple winners is an example of a *non-zero sum* game as it highlights that the win of a player is not at the expense of another, this element is what makes *non-zero sum* games non–strictly competitive as opposed to *zero sum* games that are strictly competitive [20]. Games can have both zero sum and non-zero sum elements and *Monopoly* is an example of this. *Monopoly* requires there to be one winner at the expense of the loss of others whilst still needing to cooperate with its opponents when buying and selling properties.

One, Two and N-Player Games

Games that require a finite number of players are classified as *n-player* games, where *n* denotes the maximum number of players [12]. *Monopoly* is as a two to eight player game. Although, the number of players can be a suggestion, there are games that need the set amount. For example, a two player game, like *Stratego*, requires two players. The strategy that each player takes can be affected by the number of players playing as an increase of players leads to an increase in difficulty when trying to assess and predict the moves of its opponent/s. For example, in *Monopoly*, a favoured strategy used by experienced players is to buy as many properties as soon as possible however, an increase in players decreases the chances of buying and the property distribution. Therefore, another strategy might be more appropriate in order to generate a better payoff.

For example, *The Hotelling* game, requires different strategy as the number of players increase. The concept is that the players simultaneously decide where their ice cream van is placed. Customers, are randomly placed, would go to the closest van. The winner would be the van that attracts the most customers. Now, in a two player game, the nash equilibrium is both players placing the van in the

middle, as you are guaranteed at least half of the customers. However, an increase in players makes things more difficult, particularly when there are an odd number of players as the distribution is not equal [21]. With the constraint of limited turns in the project's version of *Monopoly*, deciding when the best time to buy and sell will be crucial. Combined with the increase in players complicates things further as it leads to an increase in different strategies deployed by the opponents. Trying to align a strategy with several others so it still provides a better payoff can be difficult.

Simultaneous and Sequential Games

To classify a game as *simultaneous* each of the players adopt a strategy without any information or knowledge of the other players' strategy [22]. The decisions made by each of the players are *simultaneous*, meaning that the players decide on their move at the same time [23]. A simple example would be *Rock, Paper, Scissors*. Both players decide and display their move at the exact same moment. Each of their strategies are unknown to their opponent. It is also an example of a *complete information* game, whereby each player knows exactly how to win [23]. We know that rock beats scissors, scissors beats paper and paper beats rock. *Simultaneous* games are usually represented by payoff matrix, like below.

		Player 2		
		Rock	Paper	Scissor
Player 1	Rock	0, 0	-1, 1	1, -1
	Paper	1, -1	0, 0	-1, 1
	Scissor	-1, 1	1, -1	0, 0

Figure 2.3: Rock,Paper, Scissors Payoff Matrix

A *sequential* game requires players to make a series of moves which affects the successive possibilities of future moves [24]. As the players take it in turn, it provides the successive player information of the payoff of the previous player's moves, which can then be used to strategically plan future moves. *Monopoly* is a *sequential* game as players take it in turn to make any decisions and the successive player can capitalise on this. *Sequential* games are usually represented by *decision trees*.

Cooperative or Non-Cooperative

A *cooperative* game allows binding agreements that are followed through by the players, who benefit through cooperation [25]. It is essential that any agreements formed are enforced, otherwise it would not be classed as a *cooperative* game. Also, there needs to be a way where the payoff is distributed amongst the involved players accordingly [18]. It does necessarily have to be between only two players. For example, in *Monopoly* a trade between two players can involve the contribution of the other players. The game should consist of three players or more, otherwise creating a binding agreement would be unlikley because strategically speaking, an agreement could assist the opponent in winning. *Monopoly*

would be classed as a *cooperative* game because the main aspect of the game is to create deals with opponents, so binding agreements, in order to win the game. This element of dealing can have great impact on who wins because the distribution of payoff to all the players must be, particularly when a'good' payoff distribution provides an even better payoff for your opponent.

A *non-cooperative* game does not allow players to make binding agreements [25]. There could be a number of reasons why forming contracts and coalitions would not be possible. For example, the players could be unable to communicate with each other or players are simply not allowed to form agreements. Games such as *Chess* would be *non-cooperative*, as it is a two player game so there would be no strategic sense to cooperate with opponents.

Symmetric and Asymmetric Games

Symmetric games are those that are not influenced by the identity of the player. This means that regardless of the identity of the player the payoff of the strategy will always remain the same. In essence, this means that a player making the same moves as another should get the same payoff [26]. To determine if a game is *symmetric* it should be able to represent a player's payoff as a transpose of the other player's payoff, like in the game *Battle of the Sexes* and *Monopoly* [26]. Alternatively, in an *asymmetric* game there is usually no identical strategy sets for both players however, this is not always the case [27].

		Player 1	
		A	B
Player 2	A	1, 1	0, 3
	B	3, 0	4, 4

Figure 2.4 (a) Symmetric Game

		Player 1	
		A	B
Player 2	A	1, 2	0, 0
	B	0, 0	1, 2

Figure 2.4(b) Asymmetric Game

2.2.4 Game Representations

Game modeling requires elements required to be known and in what detail they need to be represented. There are many elements that create a game, mentioned in *Section 2.2.1*. *Cooperative* games are usually represented in the *Characteristic Function Form*, otherwise known as *Coalition Form*, whereas *non cooperative* games are represented in *extensive* and *normal form* [28].

Normal Form

Examples of *normal form* are displayed in *figure 2.4*. It describes the game as a matrix which includes information such as the players, possible moves and the payoff for each move. The payoff is determined by a function that associates a value based on all possible combinations of moves that can be made. This type of representation is typically used for *simultaneous* games, mentioned in *Chapter 2* [29].

Extensive Form

Extensive form representation includes more information than *normal form* as the order of moves, the information available and the available actions to the players at each stage is displayed [30]. Consequently, *extensive form* is used for *sequential* games, mentioned in *Chapter 2*, as it illustrates individual moves made after each succession. Its representation is a *game tree* where each node on the tree represents a state and each branch is a possible action. It is read from the top, hence why the first node represents the start of the game and the leaf nodes include the payoff of that particular combination of moves. Due to its structure one can calculate the optimal combination of moves or strategies by using backward induction. You work up the tree calculating rational moves until you reach the top of the tree.

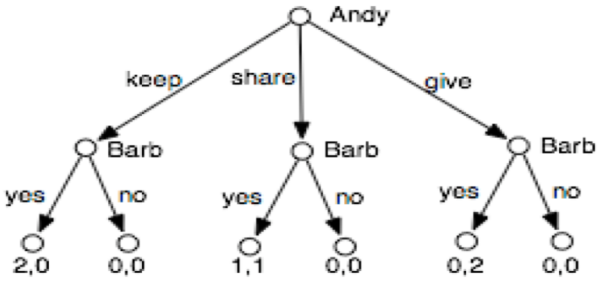

Figure 2.5: Game Tree

Coalitional Form

As mentioned above, most *cooperative* games are usually represented in the *Coalitional Form*. *Cooperative* games, mentioned in *Chapter 2*, include binding contracts between players. These contracts do not necessarily have restrictions and the possibility that they consist of ongoing payments or transfers throughout the game, which is called *transferable utility*. The inclusion of these contracts added with the objective to win comes this idea of different strategies. Many of these strategies would include players forming alliances when they share similar short term objectives. Whenever a game possesses *transferable utility*, the allocation of the payoffs in a coalition are not given separately but the coalitional form determines an overall payoff for each coalition. It includes a *characteristic function* c: $2^n \rightarrow \mathbb{R}$ which describes the payoff gained by forming a coalition.

Solved Games

For a game to be classed as a *solved* game, if both players are playing perfectly, the final payoff can be based upon the move that the player decides and the current state. The final payoff would be either a win, lose or draw. This concept is typically used for games that are *deterministic* and *perfect information*. Therefore, there is no element of luck or randomness and each player has full information. Playing 'perfectly', is when a player enforces an optimal strategy, gaining the maximum payoff possible. Examples of such *solved* games are *Connect Four* and *Checkers*.

Skilled Games

On the other hand, skilled games can never really be solved, due to the fact that they tend to be stochastic, and *imperfect information* games, mentioned in *Chapter 2*. Examples of such games are: *Poker, Blackjack* and *Risk*.

2.2.5 Game Strategies

The options that are open to a player when deciding their move, where the outcome of that option is dependent on their actions and their competitors, is what defines a player's strategy. Their strategy will influence and determine any action that the player must make at any stage of a game. Gaming Theory centralizes itself around two main types of strategies: *pure* and *mixed*.

Pure Strategy

Pure strategy is where a specific action is selected for certainty, at any decision point, without any randomization [31]. It provides a complete set of actions to take for every possible situation. Therefore, because *pure strategy* means that a set strategy is played for the entirety of the game it can be classified as *deterministic* [31]. The downsides to this type of strategy is that the opponent can begin to predict the player's move and react with an action that will grant a better payoff. For example, in *Rock-Paper-Scissors* the *pure strategy* would be to choose the same single move each time, say rock, then the opponent will be always able to predict this and retaliate by making a move that will provide a better payoff, in this case paper [32].

Mixed Strategy

Mixed strategy is where each choice in the strategy set is given a probability, where each option available is set the probability of being chosen. Therefore unlike *pure strategy* there is randomization [33][34]. As detailed in *Chapter 2*, it suggests that applying a *mixed strategy* is used for stochastic games [31]. With each choice given a positive probability, all of which summate to one, they enable to determine the player's move. Essentially, when a player is implementing a *mixed strategy* they are incorporating several pure strategies into the game [32]. This can be illustrated in the children game, *Matching Pennies* [35]. The choice set consists of two options, heads (H) or tails (T). If player 1 wins a pound from player 2 if both their choices match but would lose a pound to player 2 if they do not match [34]. This information is represented below:

		Player 2	
		Heads	Tails
Player 1	Heads	1, -1	-1, 1
	Tails	-1, 1	1, -1

Figure 2.6: Representation of Matching Pennies

2.3 Artificial Intelligence Techniques

2.3.1 Heuristic Search

The term *heuristic,* in artificial intelligence, has quite a specialized technical meaning [36]. Generally speaking, it is given to advice that is effective in most cases but not fully reliable for all cases. From an algorithmic perspective, *heuristics* would typically be used when the top priority is speed, and a satisfactory solution is acceptable and required. Consequently, completeness is sacrificed but efficiency is increased as the amount of computation is reduced [39]. The process is a mapping from a specific state to some value, through *heuristic evaluation functions*, which are analysed to find the best chance of success [36]. For example, in a *game tree*, the *heuristic* function would calculate a value for each branch level which is used to rank each branch and determine the best route to follow [37]. This value should represent the true utility of a state, without the need of doing a full search of all possible outcomes [38]. For *Monopoly*, the tree search space is very big and a full search of all possible outcomes would take a long time to compute, so a *heuristic* search would be an effective way to calculate the strategy that the AI should take.

An example *of heuristic* search would be the *Travelling Salesman Problem*. It performs a *depth first* search, creating a possible solution and verifies whether it is an actual possible solution [39]. The *heuristic* search takes the information provided about the given problem and shows which state is closer to the goal than another [40]. Using the *heuristics* calculated by the *heuristic evaluation function*, to compare and evaluate which move would provide the best payoff, helping to decide the best strategy. The *heuristic* can be designed so that it takes into consideration a game's rules and features, which can be organised into categories. For example, in *Monopoly* the rent of properties, monetary value, and opponents' need to be considered. In order to get an effective result, more information needs to be provided as it allows us to make legal moves within the limits of the game.

However, *state evaluation heuristics* requires even less computation, to calculate a *heuristic*, than heuristic search as it only takes the information of a current. It is important to note that state search can still be challenging, particularly when calculating a value for each separate component in the game. Due to this fact, the weighted linear function was introduced, a much simpler *heuristic* function:

$$\mathbf{W_1F_1 + W_2F_2 + W_3F_3 + W_4F_4 + ... + W_nF_n}$$

Where F_n denotes each component in the current state and W_n a weight parameter [38].

2.3.2 MiniMax

Minimax is a decision rule in determining the best move for the current player when the game is classified as *perfect information, sequential, zero-sum* or *deterministic*. Its algorithm is recursive and uses a *depth first search* strategy. Particularly, in a *perfect information* game, like *Chess*, it is considered that the player should never depart from a *MiniMax* strategy [41]. For *MiniMax*, its main focus is to

minimise the possible loss for the worst case scenario, so in turn maxmises on the minimum loss. When choosing a *MiniMax* strategy, the assumption made is that all players are rational [41]. It is worth noting that a *MiniMax* strategy is typically prescribed against opponents who also use a *MiniMax* strategy [41]. For a *zero sum* game, *MiniMax* is able to minimise the opponent's maximum payoff. Consequently, due to the characteristic of *zero sum* games, that a player's gain is due to the loss of an opponent, it allows *MiniMax* to monopolize on this.

MiniMax was originally created for *zero sum* games that required two players so that it could cover all the possible moves in a game, but was then utilized for *n-player* games, mentioned in *Chapter 2*. Its visual representation usually takes the form of a *game tree*, where each node is given an evaluation score. During the process of deciding the best moves for the player, the path with the highest payoff is chosen and there is an attempt to look ahead and predict what move the opponent would make, with the assumption that the opponent is playing optimally. All possible paths are looked at, starting from the root node and working downwards.

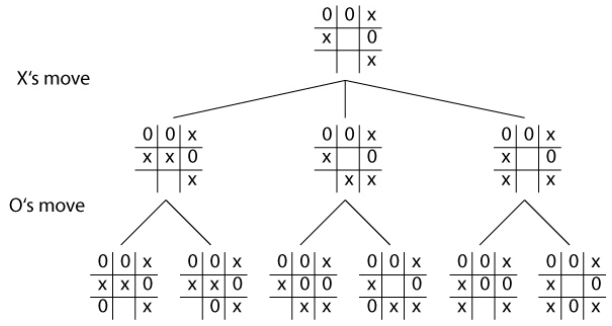

Figure 2.7: Tic – Tac – Toe Game Tree

The *heuristic* element of *minimax* occurs when each state is given its value or ranking. This ranking is provided using the *heuristic* function. Overall, it would produce a scored state board which is used to find the optimal route.

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Figure 2.8: Minimax Pseudo code

2.3.3 MaxMin

MaxMin is used when trying to maximise the minimum gain, perfect for *non-zero sum* games as it enables to maxmise the player's minimum payoff; a variant to the *MiniMax* algorithm. It is useful when the opponent is irrational and not able to play optimally [42].

2.3.4 Alpha Pruning

Both *MiniMax* and *MaxMin* algorithms use a *depth first search* technique to search all possible moves in a game, via the *game tree*. However, this can be both costly and long if the *game tree* is big. In fact, it can be exponentially expensive $O(x^n)$, where *n* denotes the maximum depth of the tree and *x* denotes the worst possible branch value. Unlike *MiniMax*, *Alpha pruning* eliminates any paths that are worse ranked than any evaluated previously [43]. Subsequently, the algorithm requires two more values to be stored. *Alpha* to store for the maximum value for the maximum path and *beta* to store for the minimum value for the minimum path. Essentially, this means that it removes or ignores (*prunes*) parts of *the game tree* that have no real impact on the goal. It works its way down the tree storing the maximum value, *alpha*, for each node and then compares this with the branch below. If the next branch is higher *alpha* is updated and the next branch is investigated, otherwise the next branch is *pruned*. The *MiniMax* element is when the *alpha* and *beta* values are passed down, as they are done with each recursive call of *MiniMax*. This same process is done for the opposition except this time instead of working with *alpha* you would be investigating the minimum path, so you would focus on *beta*. This adaptation to the *MiniMax* algorithm means the time complexity can be minimised to $O(X^{n/2})$ [38]. The *heuristic* element occurs at the leaf nodes where the *heuristic state evaluation function* is applied.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
---
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
---
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 2.9: Alpha- Beta Pseudo code

2.3.5 Machine Learning

Machine learning is a complete contrast to that of the very logic based AI techniques mentioned so far. Its main idea is to learn from experience rather than be a hard coded AI system that needs to be told how to react in all circumstances. *Machine learning* is not a method that has been used in gaming [44]. *Machine Learning* can be classified into three categories: *supervised learning*, *reinforcement learning* and *unsupervised learning*. All three differentiate based upon how the algorithm is designed to learn.

Supervised learning, is a set of examples, called a *training set*, where given an input the recommended output is provided. It provides the AI with examples of how to react appropriately and a foundation on which it can base its actions on. This set will be amended to handle any situation by producing a function that provides the appropriate output for the given input. Examples of this are *support vector machines* and *decision tree learning*.

Unsupervised learning, has three sections within itself: *k - means clustering, probabilistic clustering* and *hierarchical clustering*. It is considered to be the opposite to *supervised learning*, in the sense that *no training set* given. This means that the agent does not have anything to base its actions against. Instead, it attempts to find similarities in the inputs provided and categorises those that are similar together [45].

Reinforcement learning, is a cross between *supervised* and *unsupervised learning*. The algorithm knows an incorrect output, like in *supervised learning*, but is not aware in how to fix it, like in *unsupervised learning*. It is forced to investigate to try and figure out how to correct an incorrect answer [45]. It is not provided with a *training set* of examples on which to base its actions against, but merely told when an output is incorrect. There is no thorough guide on exactly what moves should be taken like in *supervised learning*.

For this project, a *machine learning* approach will not be taken, but a *reinforcement* approach would be the most suitable if it was. Due to the fact that we are not fully sure on what strategy would be the best so we cannot provide the all answers to all situations, like in *supervised learning. Unsupervised learning* would not be suitable too because patterns in input is not the focus.

Chapter Three

Monopoly

3.1 Introduction

There were several reasons why *Monopoly* was chosen. The most attractive was the objective of the game, which is to become the wealthiest player through buying, selling and renting properties. Determining how each player would interact in these situations generated intrigue as each player would possess a different strategy so would buy and sell at varied prices. This interaction between the players makes it worth studying and testing what would be considered the best strategy.

There are several versions of this game, with the main difference being on how a winner is declared. The original version ends the game when every player except one has declared bankruptcy, where the last player is the winner. Other versions either limit the buying and selling element, like *Monopoly Junior*, or add a time limit and declare whomever is the wealthiest at that point as the winner. To make this project as successful as possible there will be a round constraint, which mean that each player will have the same set number of turns before the game ends. Then player with the highest total, by adding up bank balances and rental property values, is the winner.

Figure 3.1 (a) game of Monopoly

Figure 3.1 (b) an example of a title deed

The rules and setup described below is for the original version of Monopoly, this project will be centered around this but there will be a few variations that are mentioned in *section 3.4*.

3.2 Setup

- Two to eight players
- Each player is given £1500
- Each player chooses a token to represent themselves on the board
- Chance and Community Chess cards are placed in the centre
- Title Deeds are given to the bank
- One of the players will also be the banker

3.3 Rules

1. Starting with the banker, each player takes it turns to roll the dice. The player with the highest roll goes first, moving their token by the number of spaces shown on the dice. The timer would be set once the banker does his first roll.

2. If a token land on a Chance or Community Chest space they must pick the appropriate card and follow the rules written on the card.

3. Once their token manages to go across the board and pass "GO" they receive £200 from the bank are allowed to start buying and selling real estate.

4. Whenever a token lands on unowned property they have the chance to buy that property from the bank at its printed price, if they choose to accept they would then receive the Title Deed card from the bank which they would place upwards in front of them.

5. If they choose to decline the banker sells it at auction to the highest bidder. The buyer is to pay the bank the amount of the bid and receives the Title Deed card from the bank. Any player, including the player who initially declined to buy at the printed price, is open to bid at auction. Bidding may start at any price.

6. If a token lands on a property that is owned by a player, they must pay the amount declared on the Title Deed card. However, if the property is mortgaged, no rent is to be collected.

7. If a player lands on "Go to Jail", the player must immediately go to jail without receiving the £200 for passing "Go". They can only leave if the pay £50 to the bank or have "Get out of Jail" card. Regardless if a player is in jail, they can still buy and sell property, buy and sell houses and hotels as well as collect rent.

8. A player can only start buying houses and hotels once they own all properties in a colour set. The price of buying a house or hotel will vary on the property, these prices are listed on the Title Deed card. A player can only buy hotels once they have bought the all houses for that property, if they decide to buy a hotel for that property they give back the four houses and keep just the hotel.

9. Players can trade among themselves as long as both players agree to the deal.

10. The game continues until all but one player declares bankruptcy.

3.4 Project Variation

This version of *Monopoly* will keep the strategic aspect of the game but will exclude less important or trivial factors out.

- Purely, to simplify the game, the element of mortgaging and the community chest and chance cards will be removed from this implementation of the game.

- Players can buy property within their first roll.
- No building allowed
- There is no £200 for passing ‘GO’ or no ‘Jail’ space on the board.
- Also, instead of playing until all but one player has gone bankrupt the game continues up until it has completed the allocated number of rounds or if one player goes bankrupt.

3.5 Classification

Knowing the types of games, as mentioned in *Chapter 2*, *Monopoly (and this projects’ version)* would be classified as:

Cooperative: The main essence of the game is to generate deals by buying and selling properties which means that contracts have to be made and enforced.

Symmetric: Identity is irrelevant when it comes down to the payoff of strategies

Sequential: Moves have to be taken sequentially, each player must wait their turn □

Stochastic: There is an element of chance, that being the dice

Perfect: All information is displayed, there is no information hidden or concealed

N- Player: Requires two or more players

Zero-Sum: The win of a player is at the expense of another

3.6 Strategies in Monopoly

There are several well-known strategies when playing Monopoly. There are a few mentioned below:

- Buy everything that you land on, adopted by Bjørn Halvard Knappskog (2009 World Champion).
- Focus on the smaller valued properties, in order to generate cash flow.
- Do not buy railways or utilities.
- Buy any property that is seven squares from any of your opponent.

- Achieve a deal where both players win but your payoff is greater.

- Buy orange and red properties as they have the highest probability of being landed on

Of course, these strategies are adopted for the full implementation of the game and so will have to be adapted for the version of *Monopoly* that will be implemented. There are strategies that contradict those mentioned above.

3.7 Approach

There are many factors that need to be considered, such as when would be the best time to buy and sell, how much properties should be bought and sold for, the issue of breaking even and making a profit before the number of turns has been achieved etc. The game will be broken into three major areas: buying, trading and auctioning. Each area will be investigated in order to find the best implementation.

The main part of this project occurs through the implementation of the AI. Different strategies require different parameters for the *heuristic evaluation function*. The opposition needs to be considered as the price of a property may vary depending on who is buying it. In order to generate a better payoff a player would not only consider their own postilion but also compare themselves to the other players in the game.

3.8 Chosen Language

The decision on which language would be used to implement this project is very crucial. Due to the finer details that could cause issues and the complexity of the game itself I wanted to relieve the pressure by choosing a language that was of higher level than languages such as C/C++. This is because it requires the programmer to be much more concise and pedantic, which is an added unnecessary problem. For example, memory has to be managed within the code which makes it more prone to bugs and errors. As *Monopoly* can be considered as a long game that has many states the *game tree* would be very large. Thus, automatic memory management is essential, influencing the decision to use Python, an object orientated language. Also, through personal experience, Python is a lot easier to express ideas.

3.9 Ethical, Social, Legal and Professional Issues

The testing phase had no inclusion of human players, all included the use of the AI players.

Therefore, there was no need for any consent, or need to respect anonymity, confidentiality or privacy. There has been no use of any external participants for this project or personal data used. Therefore, there have been no social and ethical issues in the project.

It was made a high priority that the referencing system was enforced throughout the project. Any external resources mentioned have been referenced accordingly and no information was plagiarised from the internet. The support and guidance that was received from Dr Brandon Bennett in the creation of code used throughout the project was mentioned in *Chapter 5*.

This project demonstrated professionalism throughout. It was clearly planned, realistic in its design and targets and was properly referenced.

Of course, if the project was to be developed further and there was an inclusion of human players then the above issues would be much more detailed.

Chapter Four

Design

4.1 Introduction

This chapter will describe the design of the game and the AI using the information and knowledge gathered in *Chapter 2*. In order to meet the objectives for this project, the AI techniques that will be used were explored in the background research.

Creating a two player game would mean that there is the possibility of little interaction, particularly if the AI adopts a 'no buying' strategy. However, this strategy would be interesting to test therefore, it has been decided that the game will be tested with a minimum of three players.

4.2 Game Setup

The game board will be a class in Python, which includes an array that will hold information such as the property name, property value, set and rent value. This will remain the same throughout, displaying itself as a table and will look the same for all players.

If a player owns a property, it would simply be represented by their name by that space on the table. If a player lands on an opponent's property their name can be found under the "Occupants" column.

In terms of cash, each player will be able to see the bank balance of each player as it is shown underneath the table, along with the number of properties each player owns, the total rental value of their properties and their *heuristics*. As each player takes their turn the display should output the player taking its turn, the round they are on, what they are doing, what the dice rolls, what position they move to, what their options are, the *heuristic* evaluation from the option and finally, the decision they make. For each game, the players are shuffled to decide the order of turns.

4.3 AI Design

Based upon background research the AI technique would use the *heuristic state evaluation*. The AI must be clear on which factor of the game is top priority, whether that be buying properties or retaining cash balance. Of course the decisions made will be reflected in the strategy of the AI player and in the heuristic function parameters. By creating a detailed *heuristic evaluation function,* it should be able to instruct the AI player to select the next best move. Every decision made is based upon the *heuristic*, and this *heuristic* must factor in the current state and then any changes incurred.

A *heuristic* is sufficient enough to evaluate the value of game states and to calculate the benefit of any buying or selling actions. This approach will help determine the possible moves for themselves while factoring in their opponents. The function must consider the key factors, otherwise the AI could be making short term gains and long term failures. The function will evaluate each state, taking into consideration property value, cash remaining, opposition and margins.

The Three Major Aspects of the Game: Deal, Buy and Auction Analysis

A player can only do any of these options when it is their turn.

AI Auction

All players are allowed to participate in the auction as long as they can provide the bid total if they win. Of course, this question of buying a property is only relevant if the property is unowned or if making a deal with another player. The project will be implementing the *English auction*. When a property goes for auction any players that want the property continuously provide bids, in ascending order, until there is only one player that is willing to pay the totaled bid amount. It will increment in 1's and the AI player will continue to bid based upon the *heuristic* generated.

A player would want to bid high enough that they gain the property but low enough that there is no unnecessary expenditure, which is where margins come into play. When trying to find the optimal strategy overall, for each key area of the game a different tactic may be required. Strategies usually depend on the strategy of its opponents. A technique to evaluate a strategy is using *deviation logic*.

AI Deal

A player can only trade a property that is in their possession. All trades must be beneficial to all parties involved, but the payoff does not need to be equivalent. At the beginning of each turn a player will have the opportunity to offer properties for sale which the other players can then buy. The trading aspect of *Monopoly* is instrumental in determining the winner and is a large aspect of a player's strategy. The AI would come up with a deal by calculating how much they would gain by a given transaction by comparing the value of the game state before and after the transaction is made. Trading too early can be risky because to assess a trade that will provide a maximum payoff, properties must be distributed among the players. This will then limit unnecessary expenditure. Also, when creating a trade, it is sensible to look at the opponent's assets and the repercussions of the trade, for both players. Initially, what may look like a good deal may be an even better deal for the opponent.

The trading strategy will depend on a few factors such as: player's bank balance in comparison to their opponents, the properties of their opponents and player's current position. The player's stated goal is to

maximise their payoff whilst minimizing their opponents. The added bonus of gaining property sets means that selling will be done carefully as losing a property that may need to gain a full set could be detrimental to winning.

AI Buy

The buying aspect of the AI player is embedded within the process of dealing and auctioning. As with dealing and auctioning the decision to buy a property will be based upon the rental value of the property, the player's bank balance, the opposition's bank balance and their properties. With the added incentive of rent prices doubling when gaining property sets, the last factor will have a major effect. Throughout the game, margins will be key, because they represent how much the player wants to benefit from any type of transaction. Strategically speaking, a player may not need to buy a property but it may be beneficial to do so if their opponent needs it.

Chapter Five

Implementation

Brandon Bennett helped massively with the implementation of the Monopoly and the AI, by providing the structure required to go onto the testing phase.

5.1 Displaying the State

The game board is simply a collection of strings that displays as a table, all done in display_state, displaying the information mentioned in *Chapter 4*. It gathers most of the information from the Board class which contains all but the property owners and occupants. The property owner and occupants' information is appended into the table when necessary. There is a Space class which deals with the adding and removing of occupants as they move onto and off each space. The owners of the properties will be added to the table through this class too. All personal player information can be found in the Player class.

5.2 Heuristic State Evaluation Function

The *heuristic state evaluation function* is designed so that it uses the information from its current state and player's strategy to evaluate the value of a specific state, and then picks the move that results with the highest *heuristic*. When a player is faced to make an action, say to buy a property, the *heuristic* function will take the current state and add in the effect of buying that property, returning a *heuristic evaluation*. The player will then decide on its move based upon this value. In essence, the function is set up so that it looks into the future. Everything required for the calculation of the *heuristic* are stored within the state, so it is easier to calculate new states and their *heuristics*, all that needs to be done is gathering the player's personal information. This function will be used numerous times for different occasions such as helping to calculate the buying, dealing and auctioning *heuristic* values.

Six factors that make up the *heuristic function* (Strategy class). Each factor can be changed to represent different strategies. The six factors are:

Rent multiplier (rm): In general, it is a reflection on how often a player would buy. It takes into account the rental value, property value and breaking even. For example, if a player's rental multiplier is set to 5 and a property costs £100 where the rent is £20, the player will only buy if they can get a return in 5 landings. If the rental multiplier is set to 1 then the player is highly unlikely to buy any property whereas if it is set to 8 they are much more likely.

Opponent's money multiplier (opmm): A negative multiplier of the total of opponent's money. A low value shows the player cares little about its opponents' bank balance.

Opponent's rental multiplier (oprm): A negative multiplier of the total of opponents' rent. Like above, a low value shows the player cares little about its opponents' properties.

Buy margin (bm): It represents a gain in heuristic required to buy a property. Strategically, speaking the player would want a low margin.

Sell margin (sm): It represents a gain in *heuristic* required to sell a property. Strategically, speaking the player would want a high margin. For example, if the margin is low, then a player is more likely to sell and gain a slight benefit as they would rather have the money. Both sm and bm are all about how beneficial the player wants the deal to be.

Reserve (reserve): Is the amount that the player's bank account must have at all times.

Reserve Penalty (reserve_penalty): It negates the amount from the player, if the player goes below its reserve value.

Jeopardy Aversion (ja): Is a negative multiplier of the jeopardy. Jeopardy is calculated as the fraction of spaces owned by other players; whose rent is more than the player's money. It is all about how dangerous a space is or could be for a player.

Thus, the following class was created, where the *heuristic function* calculates the *heuristic* using the rules above, the state and the player:

```
class Strategy:
      def __init__(self, rm, opmm, oprm, bm, sm, reserve, reserve_penalty, ja):
          self.rent_mult = rm
          self.opponent_money_mult = opmm
          self.opponent_rent_mult = oprm
          self.buy_margin = bm
          self.sell_margin = sm
          self.reserve = reserve
          self.reserve_penalty = reserve_penalty
          self.jeopardy_aversion = ja


       def heuristic(self, state, player):
          value = player.money(state)
          value += player.total_rent(state) * player.strategy.rent_mult
          value -= sum( [opponent.money(state) for opponent in player.opponents] ) *
self.opponent_money_mult
          value -= sum( [opponent.total_rent(state) for opponent in player.opponents] ) *
self.opponent_rent_mult


          value -= player.jeopardy(state) * self.jeopardy_aversion


          if (player.money(state) < self.reserve):
              value -= self.reserve_penalty
```

Figure 5.1: Heuristic Function

5.3 The Three Major Aspects: Buying, Dealing and Auction

Buying, dealing, auctioning, all use the information that have been stored within the sates.

Auction

The participation in an auction and the bids involved, requires the combination of several functions. Majority of the work occurs in the auction function where it takes state, space, bid and players as

parameters. It first calculates what the new state would be if the property is bought (buy_result_state) and the *heuristic* (buy_value) of buying at that bid, taking into account the strategy of the player. Also, the function considers its opponent's buying that property too (op_buy_values) and taking the *heuristic* gain for each of the players' (op_buy_states). Once it has gathered this information it will discover the minimum of these values (worst_op_buy_value) and will only continue with the auction as long as the player's *heuristic* (buy_value) is greater than its opponents. The auction function will be called within the if loop as each turn within the loop increments the property value by 1 the *heuristic function* will need to calculate how the new property value affects the player. It repeats all the calculations mentioned above, constantly checking whether an increment of 1 is still a *heuristic* gain for the player and worth bidding for. If an action is followed through then the *heuristic* gain will be added to the players existing *heuristic*, the gainFromStateChange will calculate the *heuristic* gain from the current state to the new state.

Buying

When a player rolls the dice and moves, it will get first choice on whether it wants to buy the property if it is unowned. Deciding whether to buy or not is calculated within the Progress function. It will calculate the new state including buying the property (buy_result_state) and its *heuristic* (buy_value), willing to proceed with the purchase as long as the *heuristic* is positive. It will also check the effect of an opposition buying the property and the minimum of the possible *heuristics* (worst_op_buy_value). It will decline to buy if the player's buy_value is less than the worst_op_buy_value too. It will buy if it feels that it is more damaging to let another player buy the property even if they do not need it.

Dealing

The function, highest_offer_giving_margin_gain, returns the highest offer that a buyer of a property makes to its owner and still make a *heuristic* gain for at least the given margin, within a low to high range. If a 0 payment does not make the gain margin, gm, then None is returned. If the total money, *T*, available makes the margin this will be returned and if neither of these two hold then the range is shrunk successively by evaluating the gain at the midpoint, done through the function,

highest_offer_in_range_meeting_target. The *heuristics* will then be changed depending on the outcome of the trade.

Chapter 6
AI Player Evolution

6.1 Introduction

The discovery of the optimal strategy was a methodical process. It required starting off very simple and leaving the AI nothing to consider to it slowly considering it more factors and aspects of the game. This chapter will discuss the process that took place when creating the optimal strategy for the AI player. Each of the different versions of the AI were tested and its results are shown in *Chapter 7*.

6.2 Heuristics

There will be a few *heuristic* functions, all of which have slight variation from each other, that are centred around a different top priority whilst still considering all aspects of the game. What that means is that there will be one *heuristic* that regards cash value of higher importance than buying and selling properties, one will regard buying properties of higher importance than monetary value. The *heuristic* functions will adapt and build upon each other. Initially they will be quite simple, but effective in the sense that they will capture the key essences of the game, whilst only focusing on a few elements. Slowly, this will change as the *heuristic* functions will be built upon and developed further. They will then be compared to each other in order to find the best *heuristic*. Majority of this comparison will occur during the testing stage of this project. The different *heuristic* functions and the different importance factors inherited by the AI will be tested against each other. This can then be compared against each of the players to help determine which strategy is considered the best. It is important to note that a player's strategy may only play well due to its opponents' strategy and does not suggest that it is the optimal strategy.

The testing process should generate a more balanced and well-rounded player. It will use two extreme strategies and attempt to find a balance that performs more consistently. There will be a methodical approach in trying to discover the optimal strategy. Certain strategies may initially perform well on their own but paired with others it may not. Therefore, the strategies will be tested against different strategy types to ensure the optimal is found.

6.3 AI Version 1.0

The initial approach was to simplify the AI as much as possible. This was supposed to imitate a human player, as the easiest way for a player to decide it to think irrationally. The simplest way to do this is by the AI player making moves randomly, known as random decision making, as it would not be considering any factors or aspects of the game. This process requires no *heuristics* as the measurement of different game states would be irrelevant and have no impact on the AI player's decisions. Although, this strategy method provided no reliability or assurance that the AI player would win, the same was

said for defeat too. Also, with this randomised approach it does make the AI player unpredictable which is good in the sense that its opponent cannot plan or strategically protect themselves. However, as a strategy it did not project the results required for it to be the optimal strategy, particularly when its opponents are rational.

6.4 AI Version 2.0

The advanced AI player is focused and centred around *heuristics*. Its decision is calculated and aimed to achieve its top priority. The aim of this project is to find the optimal winning strategy based on *heuristics* to be chosen for the AI player to adopt. Subsequently, the *heuristic* functions created evolved from the previous ones, taking in the things that worked and adapting the values that did not. The testing process should illustrate the AI learning from strategies. There will be four AI players playing against each other and the most winning *heuristic* parameter value will progress to the next set of testing. The AI, overall, will adopt the best *heuristic* when tested.

The *heuristic* functions could be heavily focused on certain aspects like property buying or purely selecting the moves that generated the highest initial payoff, also known as greedy decision making. In terms of greedy decision making, when applied in a game like *Monopoly*, would not consider its opponent or property sets and when it came to selling it would sell for the highest price possible and buy for the lowest. This method focuses on one aspect at a time whether that be buying for as little as possible or selling for as a high as possible, which is done by changing the multiplier.

6.5 Strategies Created

By looking at the two versions discussed above, that are focused on *greedy decision* making and *random decision* making, several basic strategies have been created, displayed in the three boundaries below. How well a strategy performs will be based on the percentage of wins within a several games. This will be the benchmark that will be used to assess how well a strategy plays. To help provide structure to the process there have been three set of parameters that have been formed, named Greedy, Tight and Irrational. These will be used as starting points. These three thresholds can be considered as boundaries between the strategies, and the optimal strategy will evolve from these.

Strategy (rm, opmm, oprm, bm, sm, reserve , reserve_penalty, ja)

Tight = Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000)

Greedy = Strategy (10, 0.5, 2, 50, 50, 0, 1200, 20000)

Irrational = Strategy (0, 0, 0, 0, 0, 0, 0)

Round 1

Strategies 1,2,3: Test three thresholds against themselves and each other.

Round 2

Strategies 4: Take the Greedy and Tight boundaries from round one and start to compare them. Taking the most winning boundary, slowly adapt its features, and the second most winning boundary. The aim is to find a balance between the two, looking at one parameter at a time. In this round it will be the rm value. Test the adapted boundary against the other to find the optimal rm value.

Round 3

Strategies 5: Taking the newly adapted boundary, search for the optimal value for opmm. This value will be take its place in the set of parameters.

Round 4

Strategies 6: Taking the newly adapted boundary, search for the optimal value for oprm. This value will be take its place in the set of parameters.

Round 5

Strategies 7: Taking the newly adapted boundary, search for the optimal value for bm. This value will be take its place in the set of parameters.

Round 6

Strategies 8: Taking the newly adapted boundary, search for the optimal value for sm. This value will be take its place in the set of parameters.

Round 7

Strategies 9: Taking the newly adapted boundary, search for the optimal value for reserve. This value will be take its place in the set of parameters.

Round 8

Strategies 11: Taking the newly adapted boundary, search for the optimal value for respen. This value will be take its place in the set of parameters.

Round 9

Strategies 13: Taking the newly adapted boundary, search for the optimal value for jep av. This value will be take its place in the set of parameters.

Final Outcome: An optimal set of parameters

Chapter 7

Testing and Evaluating

7.1 Introduction

The optimal strategy will be the strategy that wins the most and it is this strategy that will be adopted by the AI. Initially, the strategies will be tested against one other type of strategy. If no clear parameter value is found it will be tested against another type of strategy. Once the final optimal set of parameters is decided it will be tested against multiple strategies to ensure that it really is optimal.

7.2 Testing

When evaluating strategies, it is important to remember that it is assumed all participants are rational and aim to maximise their payoff within the rules of the game. Therefore, it does mean that the evaluation requires the analysis of the players and their strategic decisions which may have been influenced by the strategy of its opponents. In order to find an optimal strategy, one must consider the moves of its opponents otherwise it becomes a standard decision analysis. It must be able to handle opponents who make moves randomly and opponents who have a more structured and well thought out strategy. Due to the random element of the dice, the results may vary and 100% win rate is highly unlikely. Each strategy was tested in 1000 games, where each game possessed a maximum of 40 rounds each. It was seen through testing that 40 rounds was the most suitable as it provided the most consistent results.

NOTE: The following results display the worst cases that had been generated. Most tests were running in sets of 1000 games, where the lowest values found were displayed.

Round 1

Strategy 1 Details

Player 1,2,3,4: Strategy (0, 0, 0, 0, 0, 0, 0,0) (Irrational Player)

Player 1,2,3,4: Strategy (10, 0.5, 2, 50, 50, 0, 1200, 20000) (Greedy)

Player 1,2,3,4: Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

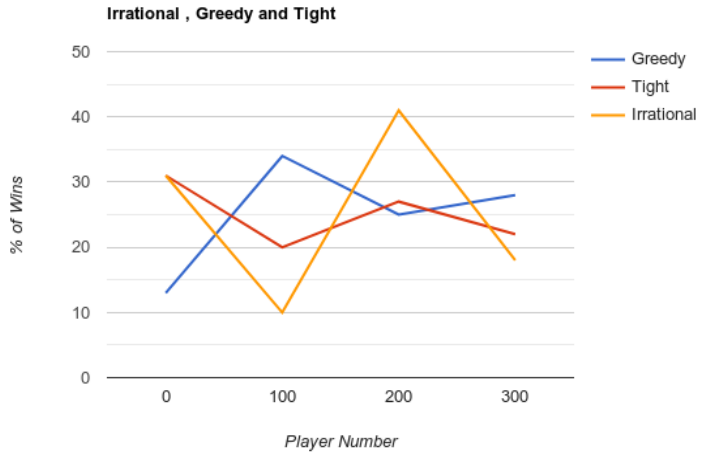

Figure 7.1: Graph showing Tight, Greedy and Irrational % of wins when playing against each other.

Strategy 2 Details

Player 1,2: Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

Player 1,2: Strategy (10, 0.5, 2, 50, 50, 0, 1200, 20000) (Greedy)

Player 3,4: Strategy (0, 0, 0, 0, 0, 0, 0) (Irrational Player)

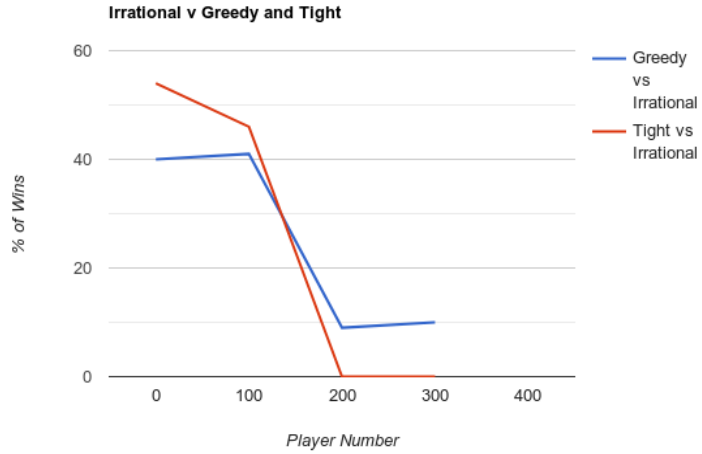

Figure 7.2: Graph showing results of Irrational Strategy players playing against Tight and Greedy players

Strategy 3 Details

Players 1,2: Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

Player 3,4: Strategy (10, 0.5, 2, 50, 50, 0, 1200, 20000) (Greedy)

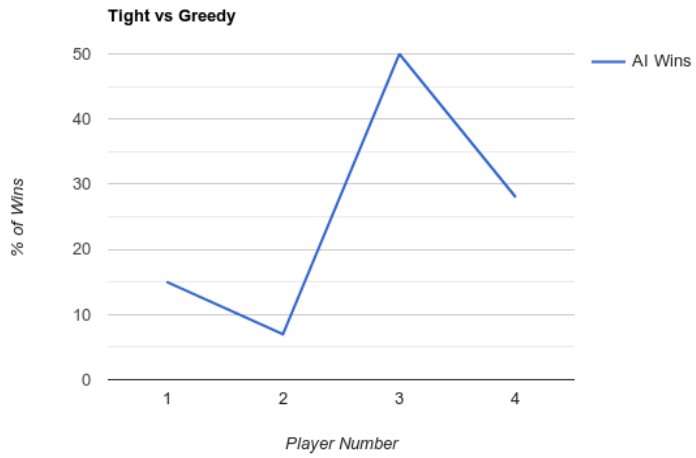

Figure 7.3: Graph showing results of Tight and Greedy players playing against each other

Outcome: Due to the unpredictability of the irrational player round 2 will not be adapting from this set of parameters. Its poor performance against rational player, displayed in *figure 7.2*, shows that discovering an optimal solution from this would be time consuming. Therefore, round 2 of testing will be focused on adapting the Tight and Greedy thresholds.

Round 2

From this round onwards the Tight and Greedy thresholds will be tested against each other, whilst slowly tweaking the parameters to find a strategy that is optimal up to this point.

Strategy 4 Details

Looking at *figure 7.3*, the Greedy strategy outperformed the Tight one. However, the difference in the percentage of wins between the two player that adapted the Greedy strategy is quite big which suggest that there is an inconsistency. Therefore, the rm parameter from the Greedy strategy will be the start of a process which will attempt to find the balance between the tight and greedy strategy in the hope that this will provide consistent optimal results.

Optimal: Strategy (rm, opmm, oprm, bm, sm, reserve , reserve_penalty, ja)

Players 1,2: Strategy (rm, 0.5, 2, 50, 50, 0, 1200, 20000) (Greedy)

Player 3,4: Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

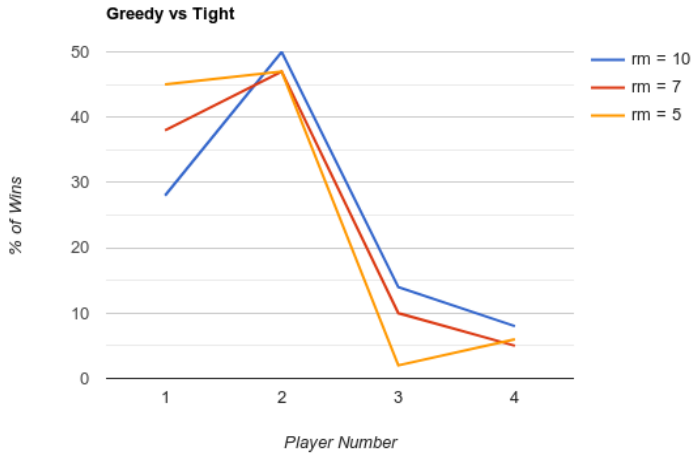

Figure 7.4: Graph showing results of Tight and Greedy players playing against each other with varied rm values

Here we consider the different levels of greediness to see if that effects the number of wins. The number of wins varied as the level of greediness changed. As the rm value decreased the consistency of the Greedy player increased whilst still generating majority number of wins against the Tight player. The optimal rm value from the graph seem to be rm=3 as it is more consistent in terms or number of wins between the greedy players.

Value for RM Taken Forward: 3

Round 3

Strategy 5 Details

Optimal: Strategy (3, opmm, oprm, bm, sm, reserve , reserve_penalty, ja)

Players 1,2: Strategy (3, oppm , 2, 50, 50, 0, 1200, 20000) (Greedy)

Player 3,4: Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

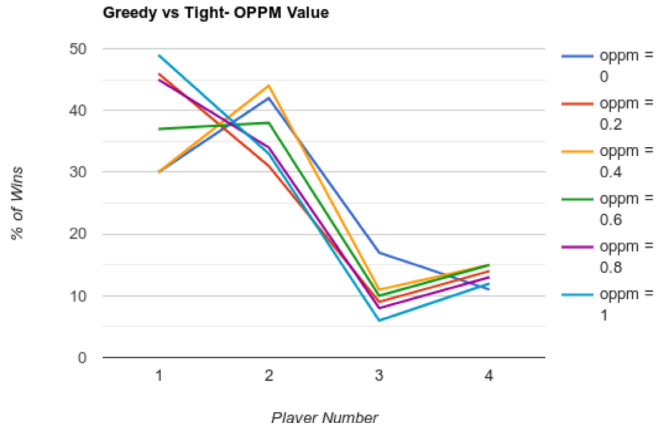

Figure 7.5: Graph showing results of Tight and Greedy players playing against each other with varied oppm values

From this set of results, we can see that a player that cares little about its opponents, when rm=0) generates less wins than players who show some or full interest in its opponents' monetary value. When the oppm is set from 0.2 to 0.6 8 the value of wins for its opponents are similar but the win distribution between the greedy players is great. From the graph above the optimal oppm value is when it is set to 1 because it allowed its opponents to win the least.

Value for OPPM Taken Forward: 0

Round 4

Strategy 6 Details

The oprm value is the same in both sets of parameters however, there may be an optimal value that is better than 2. Further tests on this parameter can be found in *Appendix C*.

Optimal: Strategy (3, 1, oprm, bm, sm, reserve , reserve_penalty, ja)

Players 1,2: Strategy (3, 0 , oprm, 50, 50, 0, 1200, 20000) (Greedy)

Player 3,4: Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

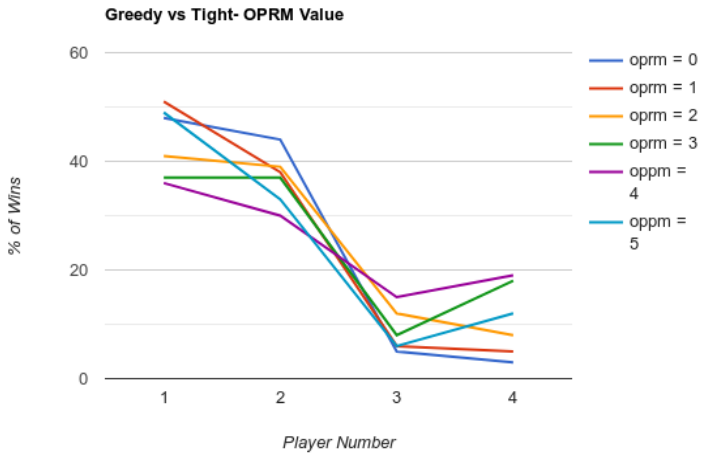

Figure 7.6: Graph showing results of Tight versus optimal players playing against each other with varied oprm values

The optimal oprm values are either 0/1/3 because they generate the most wins whilst keeping the minimal win percentage for its opponents even. To decide which value would be taken forward the set of parameters will be tested against the greedy threshold as testing it against the tight threshold did not leave much difference between the two values.

Players 1,2: Strategy (3, 1 , 0/1/3, 50, 50, 0, 1200, 20000)

Player 3,4: Strategy (10, 0.5, 2, 50, 50, 0, 1200, 20000)

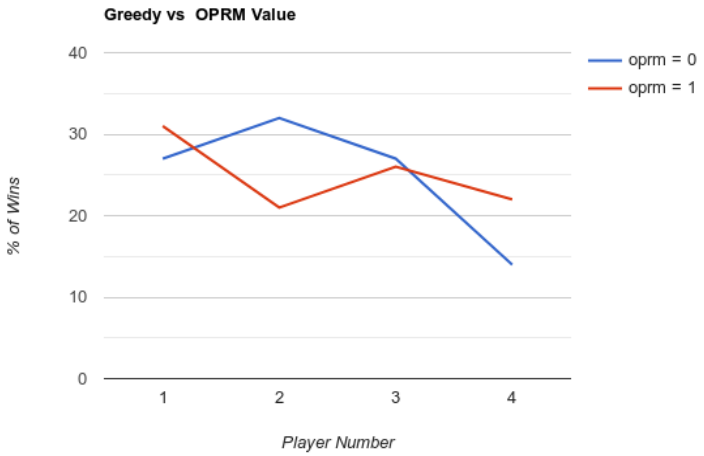

Figure 7.7: Graph showing results of Greedy players playing against Optimal players with varied oprm values

Value for OPRM Taken Forward: 1

Round 5

Strategy 7 Details

Further tests on this parameter can be found in *Appendix C*.

Optimal: Strategy (3, 0, 1, bm, sm, reserve , reserve_penalty, ja)

Players 1,2 : Strategy (3, 1 , 0, bm, 50, 0, 1200, 20000) (Greedy)

Player 3,4 : Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

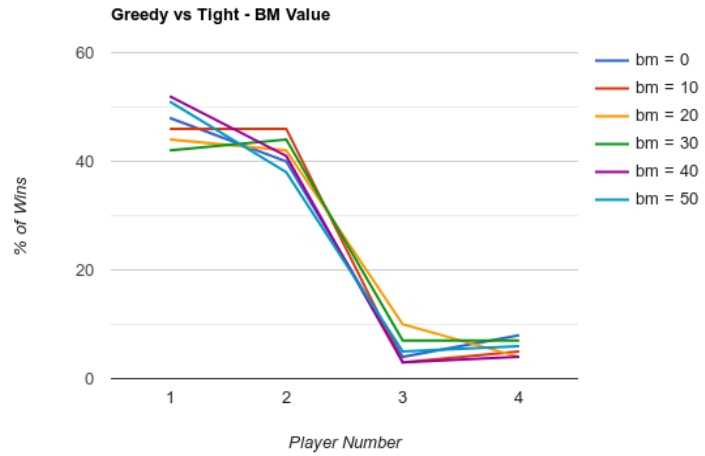

Figure 7.8: Graph showing results of Tight players playing against Optimal players with varied bm values

Looking at the graph the optimal bm value is either 10 or 50 but due to the fact that bm =10 has a more even distribution of wins whilst still keeping the opponents number of wins to a minimum.

Value for BM Taken Forward: 10

Round 6

Strategy 8 Details

Further tests on this parameter can be found in *Appendix C*.

Optimal: Strategy (3, 0, 1, 10, sm, reserve , reserve_penalty, ja)

Players 1,2 : Strategy (3, 1 , 0, 10, sm, 0, 1200, 20000) (Greedy)

Player 3,4 : Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

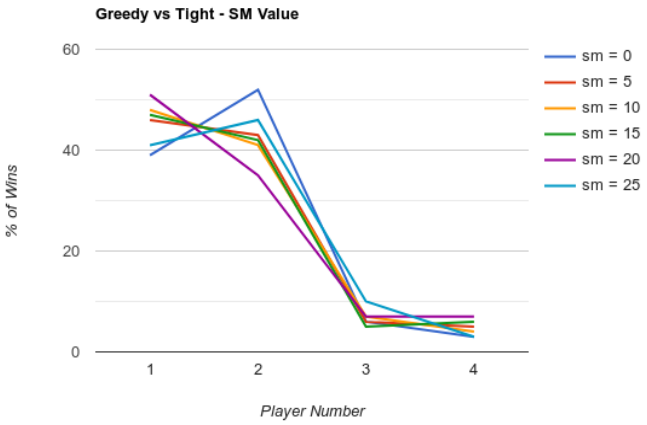

Figure 7.9: Graph showing results of Tight players playing against Optimal players with varied sm values

There is not much difference between the results form when sm=5 to sm=15. Therefore, the tests will be repeated against a tight threshold parameter.

Players 1,2 : Strategy (3, 1 , 0, 10, 5/10/15, 0, 1200, 20000)

Player 3,4 : Strategy (10, 0.5, 2, 50, 50, 0, 1200, 20000)

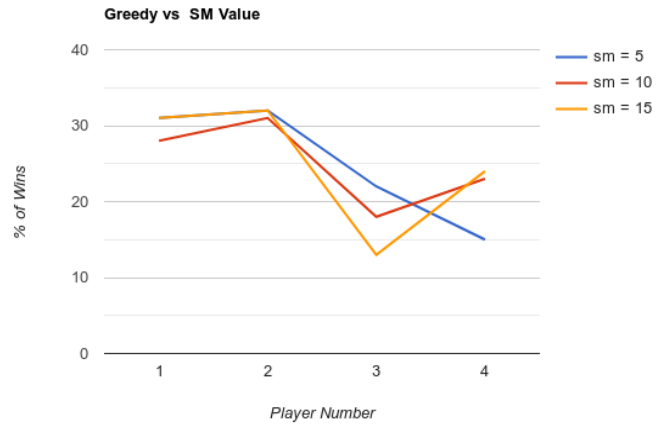

Figure 7.10: Graph showing results of Greedy players playing against Optimal players with varied sm values

The difference between the restful when sm = 10 and 15 is very small however, due to the fact that sm=10 is minutely more consistent in terms of opponent win distribution, sm=10 is the optimal value.

Value for SM Taken Forward: 10

Round 7

Strategy 9 Details

The reserve value cannot be higher than 500 as the players initially get £500 and if their reserve value is higher than they would be penalised before the game has even began.

Optimal: Strategy (3, 0, 1, 10, 10, reserve , reserve_penalty, ja)

Players 1,2 : Strategy (3, 1 , 0, 10, 10, reserve, 1200, 20000) (Greedy)

Player 3,4 : Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

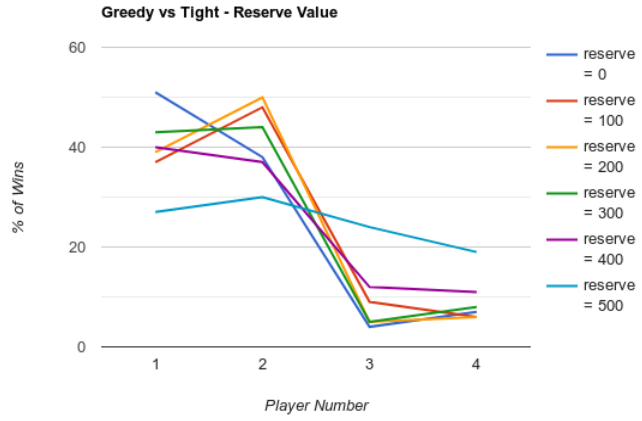

Figure 7.11: Graph showing results of Tight players playing against Optimal players with varied reserve values

The distribution of wins between each test is significantly different between each other. It is now more about which factor is more important: win distribution between the greedy players or the number of

wins by its opposing strategy. When reserve=0 is a balance between the two, its win distribution is not very even however, it generates the least wins for its opponents.

Value for RESERVE Taken Forward: 0

Round 8

Strategy 10 Details

Further tests on this parameter can be found in *Appendix C*.

Optimal: Strategy (3, 0, 1, 10, 10, 0 , reserve_penalty, ja)

Players 1,2 : Strategy (3, 1 , 0, 10, 10, 0, reserve_penalty, 20000) (Greedy)

Player 3,4 : Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

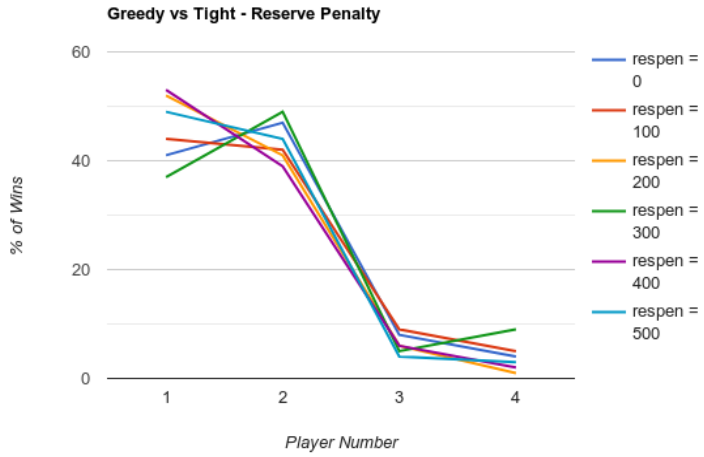

Figure 7.12: Graph showing results of Tight players playing against Optimal players with varied reserve _penalty values

The most consistent yet optimal when minimising opponent win percentage is when the reserve_penalty is set to 900. There is not much difference between the other tests however, they are not as clinical as when the reserve_penalty is set to 900. The contenders are 0, 100, 900. It was only decided that the reserve penalty would be set to 900 because of the percentage win compared to the others when playing against other types of player, test details in *Appendix C*.

Value for RESERVE PENALTY Taken Forward: 900

Round 9

Strategy 11 Details

Strategically speaking, an optimal balance between the two values would be an average of the two, so ja=1500. Further tests on this parameter can be found in appendix C.

Optimal: Strategy (3, 0, 1, 10, 10, 0 , 900, ja)

Players 1,2 : Strategy (3, 1 , 0, 10, 10, 0, 900, ja) (Greedy)

Player 3,4 : Strategy (0, 0.7, 2, 10, 5, 500, 1000, 10000) (Tight)

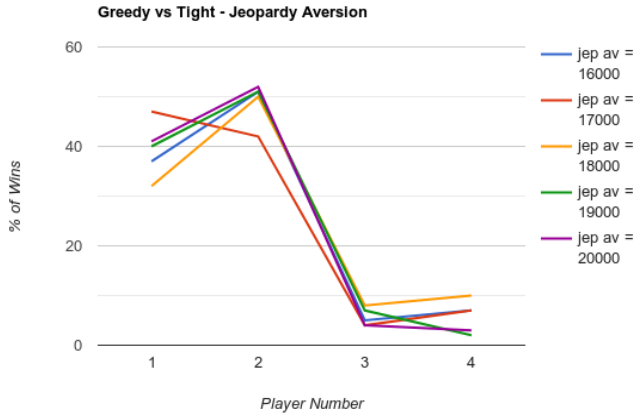

Figure 7.13: Graph showing results of Tight players playing against Optimal players with varied jeopardy aversion values

Looking at the data present in the graph when jeopardy aversion is set to 20000 that the most wins are generated for this strategy and the least wins for the opposition too.

Value for JEOPARDY AVERSION Taken Forward: 15000

Optimal: Strategy (3, 0, 1, 10, 10, 0 , 900, 20000)

Random Tester 1 = Strategy (3, 0, 3, 10, 10, 100, 500, 10000)

To ensure this is the optimal strategy it will be tested against 3 different types of strategies each time.

Strategy Type	**% of Wins**
Greedy	29
Tight	25
Optimal	46

Strategy Type	**% of Wins**
Optimal	45
Random Tester 1	39
Tight	16

Strategy Type	**% of Wins**
Optimal	53
Random Tester 1	24
Greedy	23

Figure 7.14: Three tables showing results between the optimal strategy against a range of other types of strategies.

Testing it for a range of strategies, in between the Tight and Greedy extremes as well as values that are similar to the optimal. The optimal set still beat the rest therefore; this is the final set of values for the *heuristic state function*. Further tests found in *Appendix C*.

NOTE: Testing was done on Dec-10 machines, using Spyder, purely because running the tests in 1000 game batches took a long time on my laptop.

Chapter 8

Conclusion

To conclude, I will conduct an overall evaluation of my project. This will include a verdict on whether the final implementation meets the aims of the project, a personal reflection of my own challenges and experiences and finally, an explanation on how the project could be improved and developed further.

8.1 Aim and Objectives

In order to determine of the project was a success, I will review the aims discussed in *Section 1.2* and *1.3*.

1. Investigate the different uses of AI techniques such as Minimax and Alpha-beta pruning.

A literature review was conducted to discover the different uses of AI techniques that have been used to create *Artificial Players*. were discussed and explored in *Chapter 2* of the project. From the research gathered, the AI that was implemented used a technique called *heuristic state evaluation*.

2. Develop a software that can play Monopoly in accordance to its rules

A software was developed that can play *Monopoly* with any number of players. Although, there have some aspects of the game that were not included, it still contained the strategic aspects. It was demonstrated that the game was implemented as throughout the testing phase it was played with three AI players.

3. Develop an AI algorithm to play the game of Monopoly

In *Chapter 6*, there was mentions of different strategies that an AI could implement. This included a random or greedy processes and an algorithm which incorporates *heuristic state evaluation functions*.

4. Investigate which strategies are 'better' at playing the game

In *Chapter 7* of this project, there was a methodical approach used in order to find an optimal strategy within the strategies that were tested. Particularly, the last test was the optimal strategy against several other types of strategies to ensure that the optimal strategy was found.

Overall, I feel the aim of the project has been met as there has been an AI player created and it can play in a game of *Monopoly*. The optimal strategy has shown that it can beat several types of player as its % of wins is higher than theirs and at times wins by an overall majority. Although, the game does implement all the rules the core pieces were present and the most challenging parts such as the dealing, buying and auctioning were implemented.

8.2 Personal Reflection

Throughout this project I have found experiences that were challenging however, these moments were overcome through planning and determination to succeed and make this project a success. The most challenging part was trying to understand how to create an AI player that could handle the buying, dealing and auction aspects. Dr Brandon Bennett did help in this aspect and provided a structure that went on to be used. I would say that the project has been a success as all aims have been achieved, albeit for a slightly simplified version of *Monopoly*. I have learnt from the project and it has been enjoyable most of the time. It has shown the importance of research and planning, whilst still remaining flexible.

8.3 Future Work

Monopoly

As mentioned above, this version of *Monopoly* had not included some parts such as mortgaging and community chest and chance cards. It would be nice of the game did include these things. It would be interesting to observe if these factors affected the percentage of wins for the strategies implemented.

AI Player

The AI player was pretty clever and advanced in terms of the level of the game it noticed as well as how much it considered its opponents. However, time was not factored in the heuristic which should be, particularly if rounds were going to be added. It could affect how well strategies play because an expensive purchase would not be made if there were little rounds remaining.

Methods

The AI player could be implemented using a different algorithm, as discussed in *Chapter 2*. It could be constructed through a *MiniMax* or *Alpha- beta pruning*.

List of References

[1] M. Campbell, A.J. Hoane Jr., F. Hsu (2002), *Artificial Intelligence*, page 57

[2] J. Schaffer, J. Culberson, N. Treleoar, B. Knight, P. Lu, D. Szafron (1992), *Artificial Intelligence*, page 273

[3] Buro, M. (2002). *Artificial Intelligence*. page 85.

[4] Heule, M.J.H. and Rothkrantz, L.J.M. (2007) *Solving Games: Dependence of applicable solving procedures*, Science of Computer Programming, page 1

[5] Koller, D. and Pfeffer, A. (1995) *Generating and Solving Imperfect Information Games*,Proceedings of the 14th International joint conference on Artificial intelligence, page 1185.

[6] Karlesky , M. and Voord, M.V. (2008) *Agile Project Management* (or, Burning Your Gantt Charts) , page 2.

[7] McNulty, D. (2018). *The Basics Of Game Theory*. Available at: https://www.investopedia.com/articles/financial-theory/08/game-theory-basics.asp

[8] Kockesen, L. and Ok, E. (2007). *An Introduction to Game Theory*. Available at: http://home.ku.edu.tr/~lkockesen/teaching/econ333/lectnotes/uggame.pdf

[9] Hotz, H. [no date] *A Short Introduction to Game Theory*. Available from: https://www.theorie.physik.uni-muenchen.de/lsfrey/teaching/archiv/sose_06/softmatter/talks/Heiko_Hotz-Spieltheorie-Handout.pdf.

[10] Hayes, A. (2019). *Game Theory*. Available at: http://www.investopedia.com/terms/g/gametheory.asp

[11]: Turocy, T.L. and Stengel, B. (2001). *Game Theory*. Available at: http://www.cdam.lse.ac.uk/Reports/Files/cdam-2001-09.pdf

[12]: Shoham, Y. and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. 2008: Cambridge University Press, page 121

[13]: Garratt, P. (2012) *Imperfect vs. Incomplete Information Games.* Available at: http://econ.ucsb.edu/~garratt/Econ171/Lect14_Slides.pdf (Accessed 28 Mar. 2019).

[14]: Prasad, A. (2015). *Perfect Information Vs. Complete Information*. Available at: http://bkmiba.blogspot.com/2015/09/perfect-information-vs-complete.html (Accessed 28 Mar. 2019).

[15]: Hebert, M. (2006). *Playing and Solving Games.* Available at: http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15381-s06/www/gamesI.pdf (Accessed 28 Mar. 2019).

[16]: https://ljkrakauer.com/LJK/60s/chess2.html

[17]: Nau, D. [no date]. *Introduction to Game Theory: 8. Stochastic Games*. Available at: https://www.cs.umd.edu/users/nau/game-theory/8%20Stochastic%20games.pdf (Accessed 28 Mar. 2019).

[18]: Prisner, E. (2014). *Game Theory Through Examples*. 1st edition, pages 1-2, 5.

[19]: Kenton,W. [no date] *Zero-Sum Game*. Available at: https://www.investopedia.com/terms/z/zero-sumgame.asp (Accessed 28 Mar. 2019).

[20]: Chen, J., Lu, S. and Vekhter, D. (2019). *Non-Zero-Sum Games*. Available at: https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/nonzero.html (Accessed 28 Mar. 2019).

[21]: Hotelling's Game/Median Voter Theorem with an Even Number of Competitors*(2016). Available at:* http://gametheory101.com/tag/n-player-game/ (Accessed 28 Mar. 2019).

[22]: Shor, M. (2005). *Simultaneous Game* Available at: http://www.gametheory.net/dictionary/SimultaneousGame.html (Accessed 28 Mar. 2019).

[23]: Gallego, L. (2017). *Simultaneous game* Available at: https://policonomics.com/simultaneous-game/ (Accessed 28 Mar. 2019).

[24]: Gallego, L. (2017). *Game theory III: Sequential games* Available at: https://policonomics.com/lp-game-theory3-sequential-game/ (Accessed 28 Mar. 2019).

[25]: Serna, M. (2016). *An Introduction to Cooperative Game Theory* Available at: http://www.cs.upc.edu/~mjserna/docencia/agt-miri/slides/AGT13-coop-GT.pdf (Accessed 28 Mar. 2019).

[26]: Shor, M. (2005) *Symmetric Game*. Available at: http://www.gametheory.net/dictionary/Games/SymmetricGame.html (Accessed 28 Mar. 2019).

[27]: Idurosimi (2013) *Chapter 3: Types of Games*. Available at: https://ujuzi.pressbooks.com/chapter/chapter-3-types-of-games/ (Accessed 28 Mar. 2019).

[28]: Idurosimi and Durosimi, E. (2013) Chapter *1: Representation of Games*. Available at: https://ujuzi.pressbooks.com/chapter/chapter-1/ (Accessed 28 Mar. 2019).

[29]: *Game theory I: Strategic form* Available at: https://policonomics.com/lp-game-theory1-strategic-form/ (Accessed 28 Mar. 2019).

[30]: Kockesen, L. [no date] *Game Theory: Extensive Form Games*. Available at: http://home.ku.edu.tr/~lkockesen/teaching/econ333/slides/7-Extensive-Form-Games-Handout.pdf (Accessed 28 Mar. 2019).

[31]: Dicaro, G. A. [no date]. *Game Theory 1*. Available at: https://web2.qatar.cmu.edu/~gdicaro/15382/slides/382-S18-26-GameTheory-1.pdf (Accessed 12 December 2018).

[32]: Li, V. (2016). *Pure vs. Mixed Strategies*. Available at: http://uweconsoc.com/pure-vs-mixed-strategies/ (Accessed 28 Mar. 2019).

[33]: Kockesen, L. [no date] *Game Theory: Mixed Strategy*. Available at: home.ku.edu.tr/~lkockesen/teaching/econ333/slides/4-Mixed-Strategies-Handout.pdf (Accessed 28 Mar. 2019).

[34]: Guha, M. [no date] *Mixed Strategy.* Available at: *www.columbia.edu/~rs328/MixedStrategy.pdf* (Accessed 28 Mar. 2019).

[35]: *Mixed Strategy Nash Equlibrium*. Available at: *www.econ.ohio-state.edu/jpeck/Econ5001/econ601L8.pdf* (Accessed 28 Mar. 2019).

[36]: Korf, E.R. [no date] *Heuristic Evaluation Functions in Artificial Intelligence Search Algorithms.* Available at: https://link.springer.com/content/pdf/10.1007/BF00974979.pdf (Accessed 28 Mar. 2019).

[37]: Pearl, J. (1985) *Heuristics: intelligent search strategies for computer problem solving,* pages 363-370

[38]: Russell, S. and Norvig, P. (1995). *Artificial Intelligence A Modern Approach*. 1st edition, pages 123-126, 129-136.

[39]: *Heuristic Search.* Available at: http://users.cs.cf.ac.uk/Dave.Marshall/AI2/node23.html

[40]: J, Stell, "Artificial Intelligence: Lecture 3" from <COMP 2611>. University of Leeds, Leeds, 30 January 2018.

[41]: Rapoport, A. (1999). *Two-Person Game Theory*, page 13

[tic tac toe pic]: http://theoryofprogramming.com/2017/12/12/minimax-algorithm/

[42]: Binmore, K. (2007) *Playing For Real: A Text On Game Theory*. 1st edition, page 233. □

[43]: Tarakajian,C. (2015) *Solving Tic-Tac-Toe, Part II: A Better Way. Available at:* https://catarak.github.io/blog/2015/01/07/solving-tic-tac-toe/ (Accessed 28 Mar. 2019).

[44]: Agrawal,A and Deepak, J. [no date] *When Machine Learning Meets AI and Game Theory.* Available at: http://cs229.stanford.edu/proj2012/AgrawalJaiswal-WhenMachineLearningMeetsAIandGameTheory.pdf (Accessed 29 Mar. 2019).

[45]: Marsland, S. (2014) *Machine learning: an algorithmic perspective*. 2nd edition, pages 6-9

Appendix A

External Materials

A.1 Code Repository

All of the code implemented and used in this project is available on GitLab under the following URL: https://gitlab.com/ll14m3k/ai-for-board-games-.git

Need to be logged in to access. Aq2

Appendix B

monopoly.py

```
import random
#import time


def pounds(number):
   return "£" + str(number)


class Player:
     def __init__(self, name, gender, strategy):
          self.name = name
          self.gender = gender
          self.strategy = strategy
          self.game = None
          self.games_won = 0
          self.index = None


     def display(self, state):
        game_output( "* {:<9} {:>6}   {:>2}  {:>6}   ({})".format( self.name+":",
                                    pounds(self.money(state)),
                                    len(self.properties(state)),
                                    pounds(self.total_rent(state)),
                                    int(self.heuristic(state))
                                  ) )

     def money(self,state):
        return state.money[self.index]

     def position(self,state):
        return state.positions[self.index]

     def space(self,state):
        return state.spaces[self.index]

     def properties(self,state):
        #game_output("*state.properties", state.properties)
        return state.properties[self.index]

     def heuristic(self, state):
        return self.strategy.heuristic(state, self)

     def __str__(self):
        return self.name

     def __repr__(self):
        return self.name

     def owns(self, space):
        return space in self.properties

     def total_rent(self, state):
        return sum( [prop.rent(state) for prop in self.properties(state) ] )

     ## The jeopardy of a player in a given state is the fraction of spaces on
```

```
    ## the board, which if the player landed on they would lose (because rent
    ## higher than their total money).
    def jeopardy(self, state):
        deadly = 0
        for space in self.game.board.spaces:
            if ( space.owner(state) != self
                 and
                 space.rent(state) > self.money(state)
               ):
                deadly += 1
        return deadly / len(self.game.board.spaces)




class Space:
    def __init__(self, name, cost, base_rent):
        self.name = name
        self.cost = cost
        self.base_rent = base_rent
        self.neighbours = []

    def __string__(self):
        return self.name

    def owner( self, state):
        for player in state.players:
            if self in player.properties(state):
                return player
        return None

    def occupants( self, state):
        return [ player for player in state.players
                        if state.spaces[player.index] == self]

    def display(self, state):
        self.owner(state)
        if self.owner(state) != None:
            ownerstr = self.owner(state).name
        else:
            ownerstr = ""
        if self.set_owned(state):
            doublestr = "*"
        else:
            doublestr = " "
        game_output("| {:<18} | {:>4} | {:>4}{} | {:<7} | ".format(self.name, self.cost, self.rent(state), doublestr,
ownerstr), end = "")
        occupantstr = ", ".join([str(x) for x in self.occupants(state)])
        game_output("{}".format(occupantstr))

    def add_player(self, p):
        self.occupants.append(p)

    def remove_player(self, p):
        self.occupants.remove(p)

    def set_owned(self, state):
        owner = self.owner(state)
        if owner == None:
            return False
        for p in self.neighbours:
```

```
            if p.owner(state) != owner:
              return False
        return True

    def rent(self, state):
        if self.set_owned(state):
            return self.base_rent * 2
        else:
            return self.base_rent



class GameState:
    def __init__(self):
        pass

    @staticmethod
    def startState( game ):
        gs = GameState()
        gs.game = game
        gs.board = Board()
        gs.players = game.players.copy()

        gs.positions = [0 for p in gs.players]
        gs.spaces = [gs.board.spaces[p] for p in gs.positions ]
        gs.money = [game.start_money for p in gs.players]
        gs.properties = [ [] for p in gs.players ]


        gs.round = 1
        gs.current_player_num = 0
        gs.phase = "turn start"
        gs.display = "verbose"
        return gs

    def clone( self ):
        newstate = GameState()

        newstate.game = self.game
        newstate.board = self.board
        newstate.players = self.players

        newstate.positions  = [x for x in self.positions]
        newstate.spaces     = [x for x in self.spaces]
        newstate.money      = self.money.copy()
        newstate.properties = [x.copy() for x in self.properties]

        newstate.round              = self.round
        newstate.current_player_num = self.current_player_num
        newstate.phase              = self.phase
        newstate.display            = self.display
        return newstate


    ## Calcuate heursitic gain from player going from current state to newstate.
    ## Will be negative if heuristic value goes down.
    def gainFromStateChange( self, player, newstate):
        return player.heuristic(newstate) - player.heuristic(self)

    def occupants(self, space):
        occ_list = []
```

```
        for i in range(self.game.num_players):
            if self.spaces[i] == space:
                occ_list.append(self.players[i])
        return occ_list

    def display_state(self):
        game_output("-----------------------------------------------")
        game_output("| LOCATION          | COST | RENT  | OWNER   | OCCUPANTS" )
        game_output("|-----------------------------------------------|" )
        for space in self.board.spaces:
            space.display(self)
        game_output("-----------------------------------------------")
        for player in self.players:
            player.display(self)

    def display_phase(self):
        game_output("Round: {},  Player: {},  Phase: {} ({})".format(self.round,
                                                self.current_player().name,
                                                self.phase,
                                                self.phase))

    def current_player(self):
        return self.players[self.current_player_num]

    def progress(self, display="verbose"):
        player = self.current_player()
        game_output("Phase:", self.phase)

        if self.phase == "round start":
            self.display_state()
            game_output( "Round", self.round)
            self.phase = "turn start"
            return

        if self.phase == "turn start":
            game_output( player, "to go.")
            if player.properties(self):
                self.phase = "opportunity to sell"
                return
            else:
                self.phase = "roll and move"
                return

        if self.phase == "opportunity to sell":
            game_output( player, "has the following properties for sale:")
            props_for_sale = player.properties(self)
            game_output( ", ".join([prop.name for prop in props_for_sale]))
            for prop in props_for_sale:
                offers = [(op, highest_offer_giving_margin_gain(self, op, player, prop, op.strategy.buy_margin))
                        for op in player.opponents]
                offers = [offer for offer in offers if offer[1] and offer[1] > 0]
                if offers == []:
                    game_output("No offers were made to buy {}.".format(prop.name))

                for op, offer in offers:
                    game_output( "*** {} offers £{} for {}.".format(op.name, offer, prop.name))

                offer_result_states =  [ (op, offer,
                                resultOfTrade(self, player, op, prop, offer))
                                for (op, offer) in offers ]
                 offer_result_state_vals = [ (op, offer, result_state,
```

```
                                player.heuristic(result_state))
                                for (op, offer, result_state) in offer_result_states]

            acceptable_offer_result_state_vals = [ x for x in offer_result_state_vals if x[3] >=
player.strategy.sell_margin]

            if acceptable_offer_result_state_vals == []:
               if len(offers) > 1:
                  game_output(player, "does not accept any of these offers.")
               else:
                  game_output(player, "does not accept any of this offer.")
               continue

            if len(acceptable_offer_result_state_vals) > 1:
              acceptable_offer_result_state_vals.sort(key=lambda x: x[3])
            accepted_offer =  acceptable_offer_result_state_vals[-1]
            buyer = accepted_offer[0]
            amount = accepted_offer[1]
            game_output( "DEAL: {} agrees to sell {} to {} for £{}.".format(player.name, prop.name, buyer.name,
amount ))


            #resultAllvalue = buyer.heuristic(resultAll)

         self.phase = "roll and move"
         return

       if self.phase == "roll and move":
         player = self.current_player()
         dice_num = random.randint(1,6)
         game_output( player.name, "rolls", str(dice_num)+"!" )
         self.positions[player.index] = (self.positions[player.index] + dice_num)%self.board.num_spaces
         new_space = self.board.spaces[self.positions[player.index]]
         self.spaces[player.index] =  new_space

         game_output( player.name, "moves to", new_space.name + "." )

         if new_space.cost == 0:
           game_output("This place cannot be bought.")
           self.phase = "end of turn"
           return
         if new_space.owner(self): ## someone already owns the space
           game_output("This property is owned by {}.".format(new_space.owner(self).name))
           if new_space.owner(self) == player:
             game_output("{} enjoys visiting {}.".format(player.name, new_space.name))
           else:
             game_output( "{} must pay £{} to {}.".format(player, new_space.rent(self),
new_space.owner(self).name) )
             player_money = self.money[player.index]
             if player_money < new_space.rent(self): ## Player is knocked out!
               #game_output("!!", player, "cannot pay and is knocked out of the game !!")
               self.money[new_space.owner(self).index] += player_money
               game_output( "{} gets £{} (all {}'s remaining money).".format(new_space.owner(self).name,
player_money, player.name))
               self.money[player.index] = 0
               self.phase = "bankrupcy"
               return player
             else:
               self.money[player.index] -= new_space.rent(self)
               self.money[new_space.owner(self).index] += new_space.rent(self)
           self.phase = "end of turn"
```

```
            return
        else: ## the space is available to buy
            game_output("This property is for sale for {} spondoolies.".format(new_space.cost))
            if player.money(self) < player.space(self).cost:
                game_output( player.name, "cannot afford", player.space(self).name + "." )
                self.phase = "auction"
                return
            else:
                self.phase = "opportunity to buy"
                return

    ### ---------- BUY PHASE
    if self.phase == "opportunity to buy":   # buying phase
        player = self.current_player()
        space = player.space(self)

        buy_result_state = resultOfBuy( self, player, space, space.cost)
        buy_value = player.heuristic(buy_result_state)

        op_buy_states = [ resultOfBuy( self, op, space, space.cost)
                          for op in player.opponents]
        op_buy_values = [player.heuristic(s) for s in op_buy_states]
        worst_op_buy_value = min( op_buy_values )

        #game_output(player, "evaluates current state as:", player.heuristic(self) )
        #game_output(player, "evaluates the result of buying as:", result_value )

        #gain = self.gainFromStateChange( player, buy_result_state )
        #game_output("Heuristic gain from buying:", gain)

        ## if player.money(self) < space.cost + player.strategy.reserve:
        #if gain < 0:
        if True or buy_value < worst_op_buy_value:
            game_output( player.name, "declines to buy", space.name + "." )
            self.phase = "auction"
            return
        game_output( player.name, "buys", space.name + "." )
        (self.properties[player.index]).append(space)
        self.money[player.index] -=  space.cost
        self.phase = "end of turn"
        return

    if self.phase == "auction":
        auction_space = self.spaces[player.index]
        start_index = player.index + 1
        game_output( auction_space.name, "is up for auction.")
        bid_order_players = players[start_index:].copy() + players[0:start_index].copy()
        winner, bid = auction( self, auction_space, 0, bid_order_players)
        game_output( "{} buys {} for £{}.".format(winner, auction_space.name, bid))
        (self.properties[winner.index]).append(auction_space)
        self.money[winner.index] -=  bid
        self.phase = "end of turn"
        return

    if self.phase == "end of turn":
            if display == "verbose":
                self.display_state()
            self.current_player_num += 1
            self.phase = "turn start"
            if self.current_player_num == len(self.players):
                self.current_player_num = 0
```

```
                self.round += 1
               self.phase = "round start"


def resultOfBuy(state, player, space, cost):
      newstate = state.clone()
      (newstate.properties[player.index]).append(space)
      newstate.money[player.index] -= cost
      return newstate

def resultOfTrade(state, seller, buyer, space, cost):
      newstate = state.clone()
      (newstate.properties[seller.index]).remove(space)
      (newstate.properties[buyer.index]).append(space)
      newstate.money[seller.index] += cost
      newstate.money[buyer.index] -= cost
      return newstate


def auction(state, space, bid, players):
      player = players[0]
      if len(players) == 1:
        return (player, bid)
      if player.money(state) <= bid:
         game_output( player, "passes. (Not enough money to bid)")
         return auction(state, space, bid, players[1:])
      buy_result_state = resultOfBuy( state, player, space, bid)
      buy_value = player.heuristic(buy_result_state)
      op_buy_states = [ resultOfBuy( state, op, space, bid)
                        for op in player.opponents]
      op_buy_values = [player.heuristic(s) for s in op_buy_states]
      worst_op_buy_value = min( op_buy_values )
      if buy_value > worst_op_buy_value:
        rotate = players[1:].copy()
        rotate.append(player)
        game_output( "{} bids {}.".format(player, bid+1) )
        return auction( state, space, bid+1, rotate )
      else:
        game_output( player, "passes.")
        return auction(state, space, bid, players[1:])


### This function should return the highest offer in the range low--high (L-H) that a buyer
### of space can make to its owner and make a heuristic gain for at least the given margin

### (A) If a 0 payment does not make the gain margin gm then None is returned.
### (B) If the total money T available makes the margin this will be returned
### If neither (A) nor (B) hold then successively shrink the range by evaluating
### the gain at the midpoint.

def highest_offer_giving_margin_gain(state, buyer, seller, space, margin):
        targetValue = buyer.heuristic(state) + margin
        result0 = resultOfTrade(state, seller, buyer, space, 0)
        result0value = buyer.heuristic(result0)
        if result0value < targetValue:
           return None
        allmoney = buyer.money(state)
        resultAll = resultOfTrade(state, seller, buyer, space, allmoney)
        resultAllvalue = buyer.heuristic(resultAll)
        if resultAllvalue >= targetValue:
          return allmoney
```

```
        return highest_offer_in_range_meeting_target(state, buyer, seller, space,
                                                     0, allmoney, targetValue)

def highest_offer_in_range_meeting_target(state, buyer, seller, space, low, high, targetValue):
        if low + 1 == high:
            return low
        mid = int( (low+high)/2 )
        resultMid = resultOfTrade(state, seller, buyer, space, mid)
        midValue = buyer.heuristic(resultMid)
        if midValue < targetValue:
            return highest_offer_in_range_meeting_target(state, buyer, seller, space, low, mid, targetValue)
        else:
            return highest_offer_in_range_meeting_target(state, buyer, seller, space, mid, high, targetValue)

class Game:
    def __init__(self, players, start_money, max_rounds):
        self.players = players
        self.start_money = start_money
        self.max_rounds = max_rounds
        self.num_players = len(players)

        self.state = GameState.startState(self)
        self.board = self.state.board
        for i in range(len(players)):
            players[i].game = self
            players[i].index = i
            players[i].opponents = players[i+1:].copy() + players[:i].copy()

        game_output("** ==== Leodopoly: the Leeds landlords game ==== **")

    def play(self, display="verbose"):
        self.state.display_state()

        while self.max_rounds == 0 or self.state.round <= self.max_rounds:
            result = self.state.progress(display)
            if self.state.phase == "bankrupcy":
                if result.gender == "male":
                    pronoun = "his"
                else:
                    pronoun = "her"
                game_output( result, "has lost all", pronoun, "money and is declared BANKRUPT!" )
                break

        game_output("\nThe final state of play:")
        self.state.display_state()

        if  self.state.phase == "bankrupcy":
            game_output("\n* The game ended in round {} due to bankrupcy).".format(self.state.round))
        else:
            game_output("\n* End of game (the full {} rounds have been played).".format(self.max_rounds))

        winning_amount = max([player.money(self.state) for player in players])
        winners = [player for player in players if player.money(self.state) == winning_amount]
        if len(winners) == 1:
            winner = winners[0]
            game_output( "* The winner is {} with £{}.".format(winner.name, winning_amount))
        else:
            winners_str = ", ".join([winner.name for winner in winners])
            game_output( "* The winners are {}, who have £{}.".format(winners_str,winning_amount))
        return winners
```

```
    def check_for_quit(self):
        key = input( "Press <return> to continue, or enter 'q' to quit: ")
        if key == "q" or key == "Q":
            return True
        return False




class Board:
    def __init__( self ):
        self.go =  Space("Millennium Square",   0,  0)
        ## Woodhouse
        wh1 = Space("Woodhouse Street",  60, 20)
        wh2 = Space("Melville Place",  50, 12)
        wh3 = Space("Quarry Mount",  30, 8)
        ## Hyde Park
        hp1 = Space("Hyde Park Road", 200, 60)
        hp2 = Space("Brudenell Road", 120, 40)
        hp3 = Space("Victoria Road", 250, 80)
        ## Headingley
        h1  = Space("Bearpit Gardens", 300,  2)
        h2  = Space("North Lane",    100,   50)
        ## Education
        e1 = Space("Leeds University", 700, 150)
        e2 = Space("Notre Dame", 200, 60)

        self.spaces = [
                self.go,
                wh1, wh2, wh3,
                e1,
                hp1, hp2, hp3,
                e2,
                h1, h2
               ]

        set1 = [wh1, wh2, wh3]
        set2 = [hp1, hp2, hp3]
        set3 = [h1,  h2]
        set4 = [e1,e2]
        self.sets = [set1, set2, set3, set4]
        self.num_spaces = len(self.spaces)
        for propset in self.sets:
            for prop in propset:
                #game_output(prop.name)
                prop.neighbours = propset.copy()
                prop.neighbours.remove(prop)

def test_neighbours():
    board = Board()
    for space in board.spaces:
        game_output("{}: {}".format(space.name, ", ".join([s.name for s in space.neighbours])))


class Strategy:
    def __init__(self, rm, opmm, oprm, bm, sm, reserve, reserve_penalty, ja):
        self.rent_mult = rm                # positive multiplier of total rent
        self.opponent_money_mult = opmm    # negative multiplier of total opponents' money
        self.opponent_rent_mult = oprm     # negative multiplier of total opponents' rent
```

```
        self.buy_margin = bm              # gain in heuristic required to buy property
        self.sell_margin = sm             # gain in heuristic required to sell property
        self.reserve = reserve            # minimum reserve cash
        self.reserve_penalty = reserve_penalty  # negative applied if money lower than reserve
        self.jeopardy_aversion = ja       # negative multiplier of jeopardy
        ## Jeopardy is calculated as the fraction of spaces owned by other plyers, whose
        ## rent is more than the players money.

    def heuristic(self, state, player):
        value = player.money(state)
        value += player.total_rent(state) * player.strategy.rent_mult
        #opponents = [p for p in state.players if p != player]
        value -= sum( [opponent.money(state) for opponent in player.opponents] ) * self.opponent_money_mult
        value -= sum( [opponent.total_rent(state) for opponent in player.opponents] ) * self.opponent_rent_mult

        value -= player.jeopardy(state) * self.jeopardy_aversion

        if (player.money(state) < self.reserve):
            value -= self.reserve_penalty

        return value

GAME_OUTPUT = True
def game_output(*args, end="\n"):
    if GAME_OUTPUT:
        print(*args, end=end)

# Strategies
#              rm opmm  oprm  bm   sm   res   respen  jep av
s1 =  Strategy( 0,  0.7,  2,   10,   5,  500,  1000,   10000) #TIGHT BOUNDARY
s2 =  Strategy( 3, 0,  2,   50,  50,    0,  1200,   20000) #GREEDY BOUNDARY
s3 =  Strategy( 0,   0,   0,    0,   0,   0,     0 ,   0    ) #IRRATIONAL BOUNDARY
s4 =  Strategy( 3,  0,  3,     10,  10,  100,   500,   10000) #RANDOM TESTER 1
s5 =  Strategy( 2,  0.1,  3,   10,  10,  300,   500,   10000) #RANDOM TESTER 2
s6 =  Strategy( 9,  0,    1,   10,  10,   0 ,   900,   20000) #RANDOM TESTER 3
s7 =  Strategy( 3,  0,    1,   10,  10,   0 ,   900,   20000) #OPTIMAL STRATEGY


brandon = Player("Brandon", "male",   s1 )
marya   = Player("Marya",   "female", s1 )
lucia   = Player("Lucia",   "female", s2 )
AI = Player("AI",   "female", s2 )



players = [ brandon, marya, lucia, AI]

############# Players, number of games, start money, number of rounds, game output

def test_series(players, num_games, start_money, game_length, game_output=False):
    global GAME_OUTPUT
    GAME_OUTPUT = game_output
    if GAME_OUTPUT == False:
        print("Running wihout game output ...\n")
    games_played = 0
    while games_played < num_games:
        random.shuffle(players)
        game = Game( players, start_money, game_length )
        winners = game.play("no display")
         if len(winners) == 1: ## if there is a unique winner
            winners[0].games_won += 1
```

```
            games_played += 1
    GAME_OUTPUT = True
    print("PLAYER    WINS  PERCENT |  RM  OPM  OPR   BM  SM   RES  RPEN   JEPAV")
    for player in players:
       won = player.games_won
       percent = ((won*100)/num_games)
       s = player.strategy
       print("{:<9} {:>4}    {:>5.2f}% | {:>3.1f}  {:>3.1f} {:4.1f}  {:>3} {:>3}  {:>4}  {:>4}  {:>6}"
            .format(player.name, won, percent,
                  s.rent_mult,
                  s.opponent_money_mult,s.opponent_rent_mult,
                  s.buy_margin, s.sell_margin,
                  s.reserve, s.reserve_penalty,
                  s.jeopardy_aversion ) )

test_series( players, 1000, 500, 40, game_output=False)
```

Appendix C

Heuristic Parameter Tests

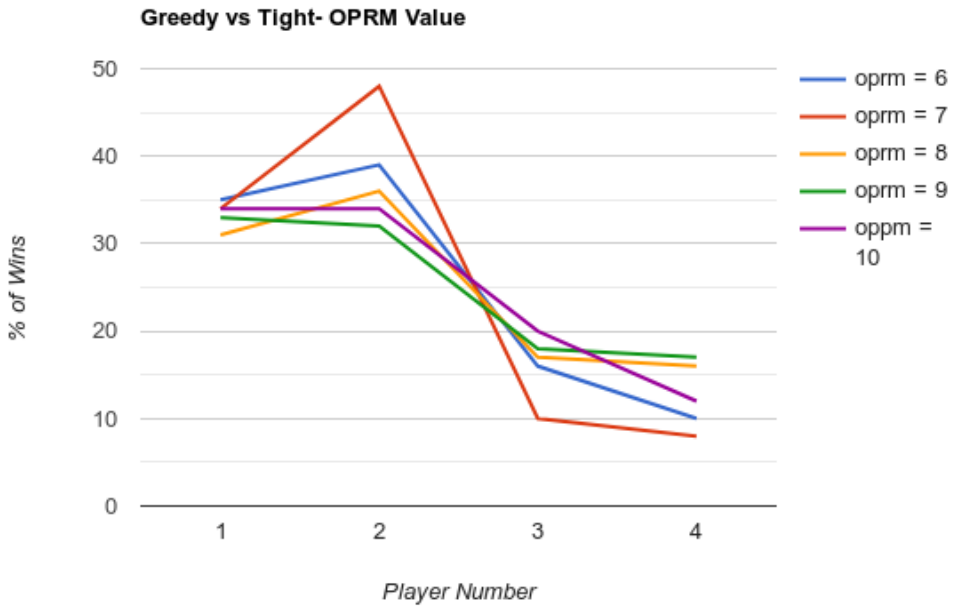

Figure C.1.1- Graph projecting the results of the oprm parameter changing from 6-10

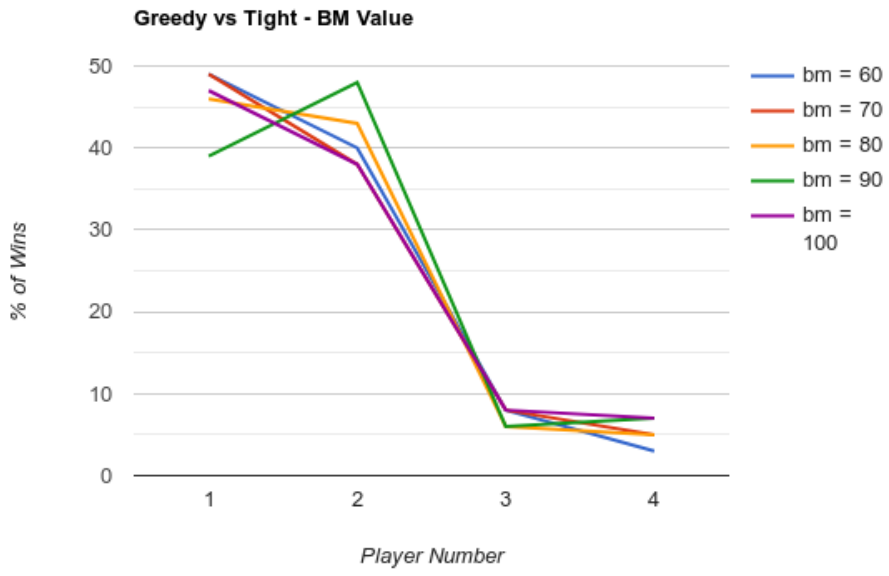

Figure C.1.2- Graph projecting the results of the bm parameter changing from 60-100

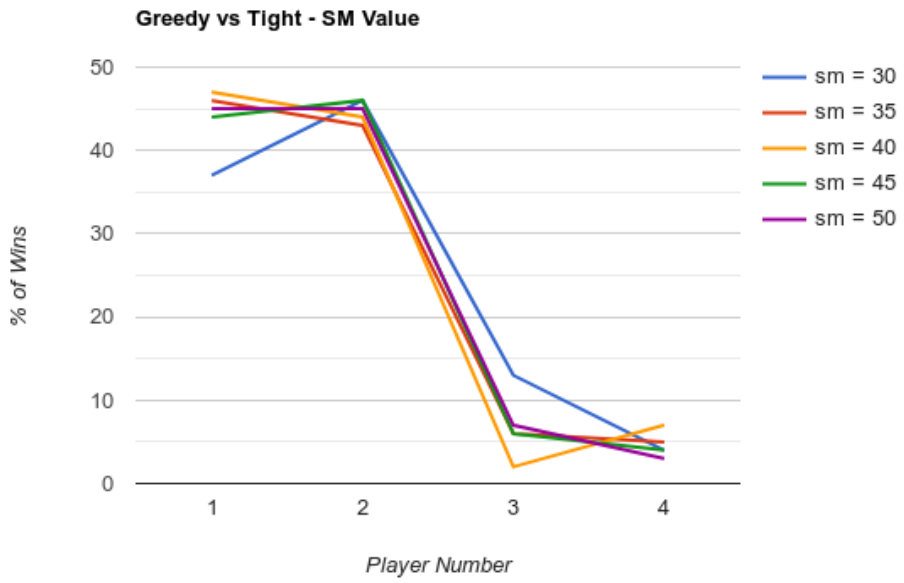

Figure C.1.3- Graph projecting the results of the sm parameter changing from 30-50

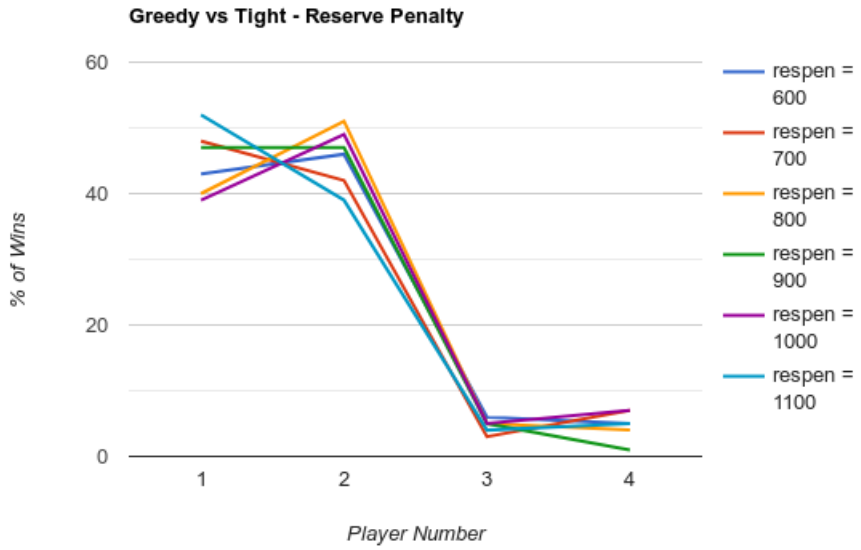

Figure C.1.4- Graph projecting the results of the reserve penalty parameter changing from 0-500

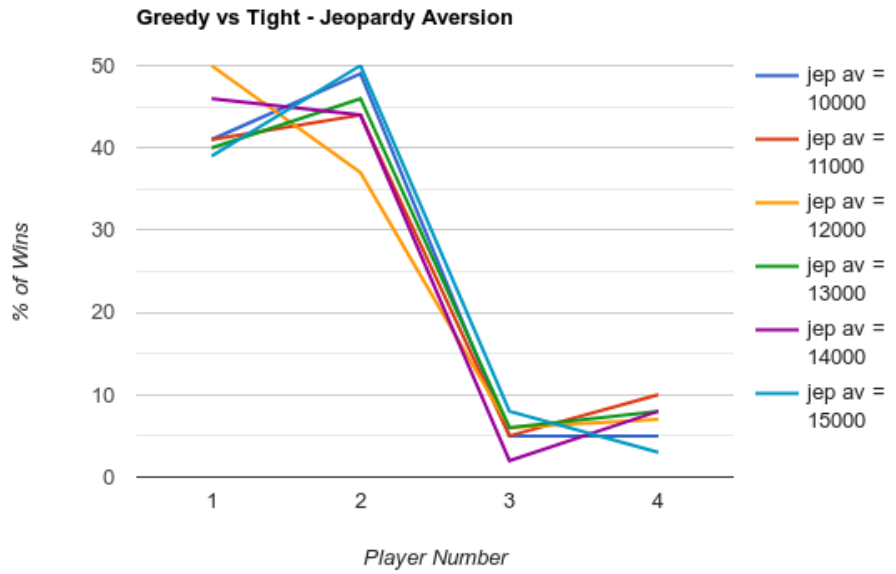

Figure C.1.5- Graph projecting the results of the jeopardy aversion parameter changing from 10000-15000

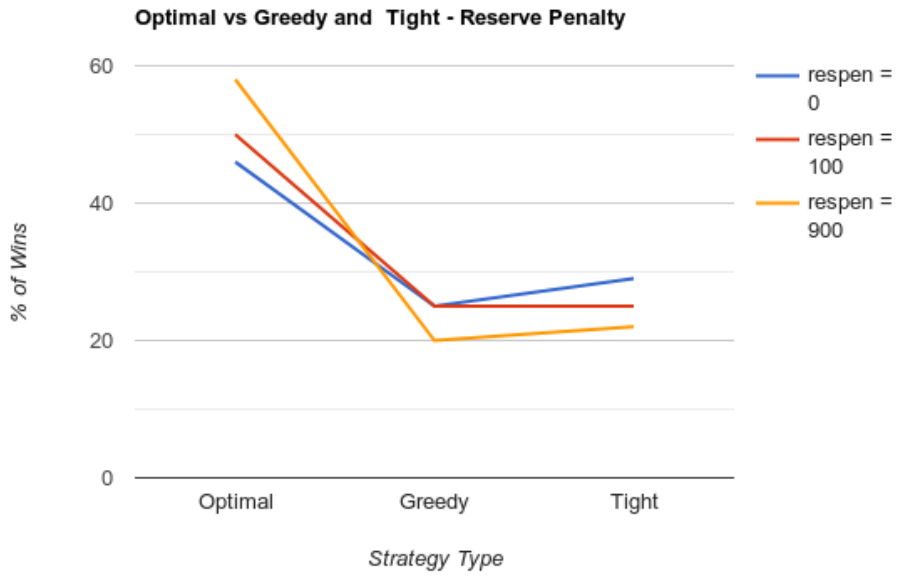

Figure C.1.5- Graph projecting the results of the reserve penalty parameter changing from 0/100/900

Strategy Type	% of Wins
Optimal	48
Random Tester 1	11
Random Tester 2	41

Figure C.1.6- Graph projecting the results of optimal set of parameters against two other strategies. Random Tester 2: (2, 0.1, 3, 10, 10, 300, 500, 10000)

Strategy Type	% of Wins
Optimal	55
Random Tester 3	20
Random Tester 1	25

Figure C.1.6- Graph projecting the results of optimal set of parameters against two other strategies. Random Tester 3: (9, 0, 1, 10, 10, 0 , 900, 20000)