School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Exploring Artificial Intelligence within Stochastic Adversarial Games involving Imperfect Information

Jordan O'Brien

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

2016/2017

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

Items	**Format**	**Recipient(s) and Date**
Project Report	*PDF*	*VLE (10/05/17)*
Project Report	*Physical Copy (x2)*	*SSO (10/05/17)*
Code	*GitLab Repository*	*Supervisor & Assessor (10/05/17)*
Participant Consent Forms	*Signed forms in envelope*	*SSO (10/05/17)*

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 explores the creation of an artificial player that has the capability of defeating human players in a relatively unknown strategy-based card game called *'Jaipur'*. The game itself exhibits properties not usually found within more well understood games such as *Chess*, therefore the methods produced have needed to consider this when making intelligent well-informed choices.

Acknowledgements

I would like to thank my supervisor Dr Brandon Bennett who gave me invaluable guidance and support throughout the entirety of my project. In addition, I would like to thank Dr Mehmet Dogar for the very helpful feedback that he provided in my progress meeting.

Table of Contents

Chapter 1 - Introduction

1.1 The Problem

Strategic adversarial games have been an ongoing source of study in relation to Artificial Intelligence since the birth of Computer Science [1][2][3][4]. Popular board games such as *Checkers*, *Connect-Four* and *Qubic* have been the focus of a lot successful research. As a result of this, techniques such as *threat-sequence searches* and *transposition tables* [5] have been used to simulate a player that can select the most beneficial move whenever one can be taken, in relation to the overall outcome of a game. However, methods such as these are only practical when the *game-theoretic values* can also be computed. This only applies to games that do not involve *chance*, *negotiation* and/or *partial/zero visibility* of the other player's cards/pieces (*imperfect information*) when the outcome is determined [5]. Therefore games that do involve these elements have received less attention.

Games such as *Poker* and *Bridge* have had less successful programs that can imitate an experienced player, due to the fact they fall under a totally different classification of game. This means that the same techniques cannot be applied to games similar to these as players will not have *complete knowledge* about the current state of the game at any one given time and moves become *non-deterministic* [6]. Therefore more research needs to be conducted in implementing and testing algorithms for artificial players within strategy games that involve both *imperfect information* and *chance*.

1.2 Project Aim

This project aims to successfully implement an artificial player capable of beating human players in a game of *'Jaipur'*. The purpose of this is to tackle the main problem (see *Section 1.1*) using a game which has characteristics that deviate from other (more traditionally studied) ones such as *Chess*. Thus, the game *'Jaipur'* has appropriately been chosen for this task.

1.3 Objectives

Initially at the beginning of the project, five main objectives were established:

1) Produce a playable implementation of the card game *'Jaipur'*.
2) Perform a background study of the techniques currently used within game playing software.
3) Develop an AI-style algorithm to play the game.
4) Test and evaluate the algorithm against human players.
5) Compare the algorithm and test results against existing studies involving other adversarial games with AI implementations.

1.4 Methodology

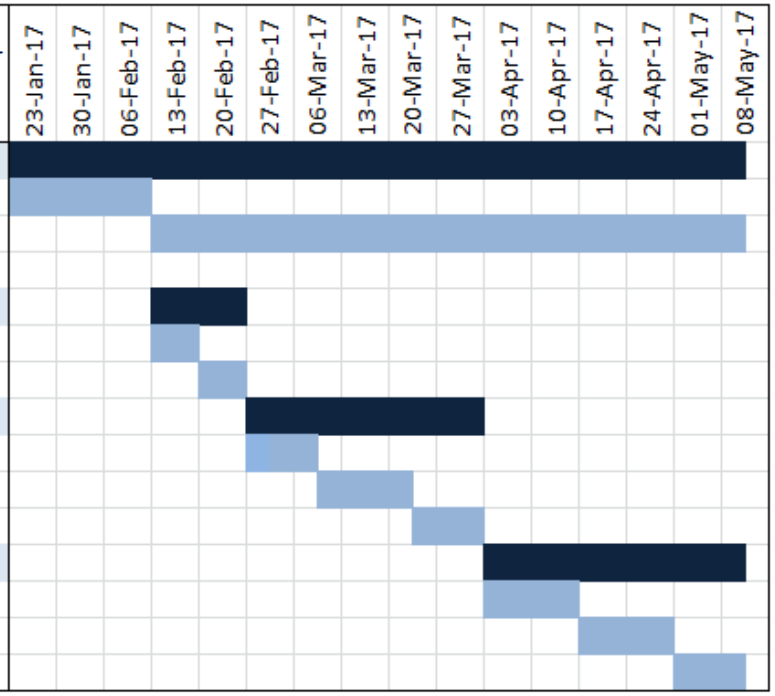
1.4.1 Initial Timeline

To shape the overall timeline of my project I initially decided to base the delivery of the main aim around my objectives. I split my project into four key stages. *Stages one*, *two* and *three* were to be completed sequentially whereas the *mandatory stage* was to be completed throughout the entirety of the project. The *mandatory stage* predominantly consisted of writing the main report. From week 3, it was decided that this was to be continuous and would coincide with the other three stages to try and avoid backlog in the days leading to the deadline.

Each stage focused on a different aspect of the project; however each one needed to be completed in order to advance to the next one. To avoid falling behind with work, I decided to take an agile-type of approach when implementing code and completing each stage. Both stages that involved developing code had two different versions: *basic* and *main*. The *basic solution* would be essential in order to progress further, and the *main solution* was to be the ideal version used within in the final report. Due to this, if I fell behind implementing one of the *basic solutions*, there would still be time available in the period reserved for creating the *main solution*. Weekly meetings with my supervisor were held at the beginning of each week to discuss the progress I had been making and to capture any potential problems. At the end of each stage, a review was held to establish and track the main progress of the project.

Every stage contained specific tasks and were subjected to individual deadlines, as summarised within the following Gantt chart.

Tasks	Start Date	End Date	Number of Days
Mandatory Stage	**23-01-16**	**10-05-16**	**108**
Scoping and Planning Document	23-01-16	12-02-16	21
Report Writing	13-02-16	10-05-16	87
Stage One	**13-02-16**	**24-02-16**	**14**
Create a playable implementation of 'Jaipur' (Basic)	13-02-16	19-02-16	7
Create a playable implementation of 'Jaipur' (Main)	20-02-16	26-02-16	7
Stage Two	**27-02-16**	**02-04-16**	**35**
Perform Background Research	27-02-16	07-03-16	9
Develop an AI-style algorithm for the game (Basic)	08-03-16	21-03-16	14
Develop an AI-style algorithm for the game (Main)	22-03-16	02-04-16	12
Stage Three	**03-04-16**	**10-05-16**	**38**
Evaluate and Test with Human Players	03-04-16	16-04-16	14
Compare Solution with Similar Studies	17-04-16	30-04-16	14
Final Report Completion	01-05-16	10-05-16	10



Below I have summarised every *basic* and *main solution* listed above:

- **Create a playable implementation of '*Jaipur*'**
  - **Basic:** Produce a platform for '*Jaipur*' which can be used by two human players through a text-based interface.
  - **Main:** Implement an interactive graphical user interface to replace the existing one.

- **Develop an AI-style algorithm for the game**
  - **Basic:** Produce one algorithm that is capable of playing '*Jaipur*' at a reasonable level of difficulty.
  - **Main**: Extend the AI solution by including additional algorithms that try and tackle the problem from a different approach.

1.4.2 Revised Timeline

As a general guide, the Gantt chart was able to model the full project scope quite well. By splitting the main work into three separate stages, it allowed me to keep relatively on track and plan each task accordingly. Creating *basic* and *main* solutions also proved to be very practical as debugging unperceivable problems became a common practice whilst coding. This would often slow down production and therefore a contingency plan would become a very desirable method to employ (later discussed in *Section 3.4*).

However, alterations had to be made in order to fully complete the background reading. Initially, I allowed myself a total of 9 days to perform the necessary research needed to implement the artificial player for the game. This was shown to be a naively optimistic amount of time and later the task had to be continued in parallel when creating the playable '*Jaipur*' platform.

Also, whilst completing each main task, trying to write the report in unison transpired to be a lot more challenging than previously anticipated. As a result, report writing became a low priority and a lot of sections were left unwritten within the first two stages. As a consequence, the final 4 weeks consisted only of finishing the main body of the report and no time was left to perform a comparison study with other similar AI solutions. This being said, I feel the Gantt chart worked reasonably well by laying out a simple plan to be used as a rough guide over the full 12 weeks.

Chapter 2 – Background Research

In order to understand how to approach the main aim of the project, I will first need to classify the problem and then decide what methods can be used to solve it. To do this, I have conducted an in depth literature review on both general *Game Theory* and *AI techniques* currently used within game playing software. After this, I have provided an insightful overview of the game '*Jaipur*' and discussed what approaches will be used to create the artificial player.

2.1 Game Theory

Game Theory is the study of strategic decision making in situations where the outcomes (and rewards) of the contributors involved rely heavily on the choices made by other contributors and themselves. Though originally designed to tackle traditional board games such as *Chess*, it has since been abstracted and now has made many advancements in numerous academic fields such as *politics* [7][8], *biology* [9] and *economics* [10][11].

2.1.1 Terminology

In order to study the different types of games, we need to first define what a game is and what the components of a game are. To do this I will be using the definitions from a modern text [12] which takes its ideas directly from important contributors in game theory such as *John Von Neumann*, *Oskar Morgenstern* and *John Nash*.

- A *Game* is described by its own set of rules.
- A *Play* is an instance of a *game*.
- A *State* is a unique configuration of all the existing components within a *game*.
- A *Move* is a decision in an individual *state*.
- A *Strategy* is a plan that tells the player what move to choose in every possible *state*.
- An *Outcome* is the consequence of a *move*.
- A *Payoff* is the reward produced by the *outcome*.
- A *Rational Player* is one that has preferences, belief about the world (including other players) and will try to optimise their individual *payoffs*.

2.1.2 Game Strategies

In game theory, two main types of strategy exist: *pure* and *mixed*. In a *pure strategy*, the player is required to choose one move at probability 1 and the remaining moves at probability 0. In contrast, a *mixed strategy* will incorporate random move selection where at least two choices have a positive probability and all probabilities summate to 1 (e.g. $p_1 = 0.7$ and $p_2 = 0.3$) [13]. Thus, a *mixed strategy* is the assignment of probability to moves belonging to separate *pure strategies*.

Mixed strategies can be used effectively in games where a predictable *pure strategy* can be of great advantage to the opponent. For example, if playing a simple game such as *Rock-Paper-Scissors* the *pure strategy* chosen always selects the move *Rock*, then the opponent would quickly learn to select the move *Paper* to earn the maximum payoff. However if they were to select either *Rock*, *Paper* or *Scissors* at equal probabilities ($p = 0.33$), the player would become desirably more unpredictable.

Games involving *chance* can also benefit in using *mixed strategies* as the probabilities distributed among move choices can reflect on real elements of the game (e.g. the likelihood of the next card in a standard shuffled deck belonging to the *suit of clubs*). It is also worth mentioning the *strategy type* chosen may depend on the number of moves available to pick from in a game throughout, this being a *fixed* or *non-fixed* value. Games with a *non-fixed* number of moves can often have a different output in different states of the game even whilst using the same *pure strategy* throughout.

2.1.3 Game Types

Zero Sum and Non-Zero Sum Games

Zero sum games have the property that the sum of the payoffs to all players is equal to zero. Thus a player can only have a positive payoff if at least one other player within the game has a negative one [12]. Consider *Chess* and *Checkers* as examples where the overall payoffs are determined by the victory status of each player. If a player wins they receive the payoff of 1, if they lose the payoff of -1 and 0 if the *game* is a draw. As only one player can win (consequently meaning the other loses), the sum of both player's overall payoffs will always be zero. Therefore it can be said the payoff of player two is the negative payoff of player one [14].

Alternatively, in a game where one player's gain does not have to come at the expense of another player, it is classified as *non-zero sum* [15]. Because of this condition, games that

can contain more than one winner at the end are typically categorised under this domain (given the *overall payoff* is determined by the victory status of each player).

One, Two and N-Player Games

The number of players in a game can greatly influence a strategy one can take to try and reach the overall objective (the objective usually being to win). Games that only contain *one player* tend to form simpler *strategies* when deciding what move to take next due to the lack of outside parties (other players) hindering one's own success. Because of this, the *strategy* chosen often tries to maximise the chances of the player gaining victory however if game offers *uncertain* elements (e.g. taking an unknown card from a shuffled deck) this should also be considered.

Two-player games are often adversarial and only focus on competing against one another to win. This being said, players must also factor in their opponent when selecting a move as it may have more payoff for them in the future than the immediate value it gives.

Games that consist of *three or more players* are commonly referred to having *n-players*. Where in *two-player zero sum* games the payoff is always the opposite of the other player, this is not always the case with more players involved. The payoffs distributed do not have to be equal however still need to respect the *zero sum property* [16]. For instance, if after a move the payoffs for players one, two and three are 7, -3 and -4 respectively, it is still valid.

Perfect and Imperfect Information Games

A game has *perfect information* if every player has a full awareness of all current and previous states for all other players. This means that every player will be aware of the previous choices of all other players in any point of a single game [17]. It should be noted that *perfect information* games are not the same as *complete information* games. These games alternatively describe situations where players have *common knowledge* of the game being played. This could include knowing what moves are possible and what the payoffs are of specific moves taken given a certain state [18].

Imperfect information occurs frequently in games where players are unable to see their opponent's cards, pieces, etc. initially and/or throughout a *play*. Good examples include *Poker*, *Scrabble* and *Stratego*.

Deterministic and Stochastic Games

A game is *deterministic* only if when a specific move is performed in a specific state of an overall game, the output is always the same no matter how many times it is executed. Every move performed in a *deterministic* game has no variation and each output is uniquely *pre-determined* given the current state and the move selected. It should also be mentioned that games that involve *imperfect information* can still be *deterministic* as even if a player cannot see the outcome of their move, it may still be determined (e.g. selecting a coordinate to "hit" in the board game *Battleship*).

If a game contains elements of *chance* such as dice rolls or card shuffling it is classified as *stochastic*. This occurs if the outcome of any action in any possible state is uncertain. One way to calculate the payoffs in stochastic games is to take weighted averages of the probabilities of the next possible states [19], thus generating a sensible estimate.

Cooperative and Non- Cooperative Games

A game is said to be *cooperative* when the results of a *negotiation* between two players can be put into a contract (a formal agreement) and enforced [12]. Players will be able to form alliances with current opponents or may be able to make an agreement that does not have to be self-enforcing [20].This means if an agreement that affects two players (e.g. a trade) is made it can have one or more other players contributing towards it.

As *cooperative games* centre on forming agreements with at least one other player, it is implied that each game of this type can have no less than three players involved. In a two-player game it would typically make no sense to try and assist the opponent winning if the objective was to beat them.

Forming contracts with opponents often can be more advantageous and mutually beneficial than working alone. The board game *Risk* often encompasses this idea, as players may find working together can quickly strengthen their current position whilst gradually eliminating players outside the formal coalition made. Also, it is suggested a necessary condition for a formation of any agreement is that it is stable, meaning members of the agreement must not have any incentive to walk away from it whilst it is effective [21].

Simultaneous and Sequential Games

Simultaneous games occur when, in a *play*, all players have only one move and all moves are made instantaneously. Alternatively, a *sequential game* does not allow players to take a move at the same time and players may be allowed to move several times before their turn

has ended [12]. Because of this, sequential games allow players to have knowledge of their opponents past moves before they select their own which can have a much greater influence on their choice. By definition, if a game is simultaneous it will also have imperfect information as the player will always be unaware of their opponent's current choice(s). The games *Rock-Paper-Scissors* and *Uno* are simultaneous as no turns are taken when selecting a move.

2.1.4 Game Representations

How a game is represented relies exclusively on whether it is classified as *simultaneous* or not. If it is then the game can be denoted in *Normal Form*. This consists of a finite set of players, a set of all possible moves assigned to each player and a payoff function which allocates a payoff to each player depending on what move was selected from each one [22]. This information can then be represented in a matrix of n-dimensions, where *n = number of players* (see *Figure 2.1*).

For *sequential* games, *Extensive Form* is used instead as it incorporates moves that are executed in succession. A game tree is used to represent the entire game where each node represents a specific state and each arc (branch) is a possible choice made by one of the players. The initial node (root) is usually the start of the game but can be the current state if the game has already begun. Nodes only connected to one arc are the end nodes (leaves). These contain the payoffs which the players receive if the combinations of move choices to get there from the initial node are selected (see *Figure 2.2*).

	Rock	Paper	Scissors
Rock	$0, 0$	$-1, 1$	$1, -1$
Paper	$1, -1$	$0, 0$	$-1, 1$
Scissors	$-1, 1$	$1, -1$	$0, 0$

Figure 2.1 – The game Rock-Paper-Scissors in Normal Form [15]

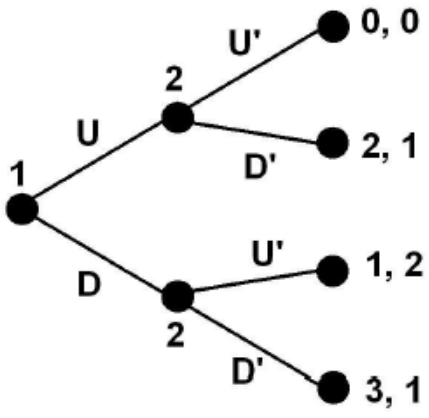


Figure 2.2 – An abstract game in Extensive Form [22]

2.1.5 "Solved" Games

Some games have the possibility of being "*solved*" which means the overall output (win, lose or draw) can be correctly predicted given the current state of the game and which move the current player selects (given the other player is also playing perfectly). *Checkers* is the most complex game solved to date; with 10^{20} different possible board positions and 10^{14} calculations to complete the proof, it took 18 years to complete [23]. However this is only possible for *deterministic* games with *perfect information*.

It can also be mentioned in more recent years this has become more achievable to accomplish due to the growth of *parallel computation*. Whereas in the past researchers have been able to solely rely on steady processor clock speed increases to tackle games with larger state spaces, this is no longer wise as clock speed increments are slowly coming to a halt [24].

2.1.6 "Skill" Games

Games such as *Poker, Bridge, Blackjack, Scrabble* and *Risk* can never be fully solved due their elements of *chance* and *imperfect information*. This has led to debates whether a game such as *Poker* can be considered a *skill game* or not for which there is no definitive answer. It can however be argued that the classification of *Poker* as a skill game does not depend on the game but on other players *behaviour* (rational /irrational) [25]. Therefore to optimise one's own moves in a game such as this, predicting the other player's strategy could potentially be the key to success.

2.2 Artificial Intelligence Techniques

2.2.1 Heuristic State Evaluation

Traditionally, heuristics are used in algorithms to find approximations to solutions when speed is a greater priority than total accuracy. This idea can be used to estimate the value of a specific state within a game by using *heuristic evaluation functions*. These provide estimates of the overall payoff when selecting a specific move in a play. If designed correctly, each function should accurately predict the true utility of a state without doing a complete search of all possible outcomes [26].

When constructing a function for a game, it should be based on the general structure and components involved. A good example is *Chess* as each piece can be designated a value (e.g. every *pawn* is worth 1, every *knight* is worth 6, every *queen* is worth 14 and so on), and depending on the positions, mobility potential and number remaining for each piece, a *heuristic evaluation function* can be used to assess all possible next moves. These evaluations are not completely precise; the player is more likely to gain better evaluations when they provide the function with more information about the current state of the game [27]. It can also be argued that an *admissible heuristic* (one that provides a lower bound estimate than the actual end state) will be more likely found if this approach is taken.

In many early game-playing programs, functions such as these were widely used as they were mainly knowledge-based. This meant that they did not rely on search but on the information available about the current state of the game, as computational speed was usually an issue [28]. One of the most common function types used within games is the *weighted linear function*:

$$w_0 f_0 + w_1 f_1 + w_2 f_2 + ... + w_n f_n$$

Each *feature* (f_0, f_1, f_2, … f_n) represents a component of the current state of the game e.g. piece/card type, number of moves remaining etc. and each *weight* (w_0, w_1, w_2, … w_n) is an *adjustable parameter* that represents the current value of each component [26].

2.2.2 Minimax

Games that incorporate classic characteristics (*perfect information*, *deterministic moves*, *sequential gameplay*, *zero-sum* and *two players*) can often use an algorithm called *minimax* which is designed to find the best move for the current player to take.

Each game is modelled in *extensive form* and the whole *game tree* is generated, displaying all possible alternating moves between players. A *utility function* is applied to all end nodes to decide if the current player wins the game in each instance. Each value is then recursively passed up the layers of the tree, finding the maximum value (*max nodes*) for when it is the current player's turn and the minimum value (*min nodes*) when it is the opposing player's. When the values eventually reach the first level of the tree, the move with the maximum payoff is selected. This is called the *minimax decision* as it maximises the payoff under the presumption the opponent is trying to minimise it [26].

```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
      VALUE[op] — MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return  UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Figure 2.3 – A typical pseudocode representation of the minimax algorithm [26]

Maximin

Variants of the *minimax algorithm* have also been designed to tackle games with alternative goals. One example is the *maximin algorithm* which aims to select the move that minimises the total payoff for the current player. As well as games where the overall objective is to lose or gain minimum points (such as the popular card game *Hearts*), it may be advantageous to use the algorithm when there is knowledge that the other player will play poorly or make ill-informed choices [29].

Alpha-Beta Pruning

Searching a full game tree using a *depth-first* process can be exponentially expensive such that the time complexity becomes $O(b^m)$ where b is the worst-case branch number and m is the maximum depth. Because of this, *pruning* was developed to disregard sections of the *game tree* that make no impact to the final choice. This method is called *alpha-beta pruning* where, if done perfectly, can reduce the overall complexity to $O(b^{m/2})$ [26].

This is done by storing the current maximum value for each node that represents the current player's turn (*alpha values*).These values are then compared to each branch value from the next level down, thus comparing all possible next moves. If the branch is lower than the

current alpha value, it is ignored (*pruned*), else the alpha value is updated and the branch is explored. The same is done for each node representing the opposition's moves using *beta values* and only exploring lower branches if the value given for them is higher.

Expectiminimax

Minimax cannot be efficiently used in games that are defined as *stochastic* as their non-deterministic elements do not guarantee the overall payoffs discovered by fully searching the game tree. Because of this, a modified version of the algorithm called *expectiminimax* was invented to factor in the probabilities of certain outcomes happening in the future. It does this by incorporating *chance nodes* between each *min* and *max* node layer of the tree, and each node value is adjusted accordingly to correspond with the chances of each outcome at each layer of the tree. However, the rest of the algorithm still operates in the same manner as in the *original minimax* [30].

As the additional *chance nodes* create a larger game tree in terms of complexity, further measures can be made. As well as using *alpha-beta pruning* to fully ignore certain branches of the game tree, *gamma pruning* can also be implemented within the algorithm to ignore chance nodes that contain probabilities under a certain threshold (such as $p = 0.05$) [31]. Existing studies have also shown that by using parallelism to enhance the performance of *expectiminimax*, the next move is chosen faster 90% of the time [32].

Cutoff Testing

In addition to *pruning*, minimax algorithms can abide by a *depth limit* when searching for the end nodes in each level of a game tree. This is called a *cutoff test*. Each node of the game tree is explored until the depth limit is reached. Once this happens, the *utility function* is replaced by a *heuristic evaluation function* and an estimate of the *ultimate payoff* (final score, game victory/defeat etc.) is calculated. These values are then fed up the tree and the algorithm continues as per usual. It should be noted however that this approach can have catastrophic consequences due to the fact the evaluation function is estimating the overall outcome [26].

The method is much more suited for *stochastic* games that are using techniques such as *expectiminimax*. This is because the look ahead of potential future states becomes more and more obscure depending on the depth of the game tree mainly due to the factors of probability involved. Therefore cutting the search earlier can potentially result in a better informed decision (given a sufficient evaluation function is used) rather than exploring every

possibility. This being said, *deterministic* games have been proven to use this method successfully such as the *Chess* playing program '*Deep Blue'* that used a depth limit of 12 before conducting a *heuristic evaluation* [33].

2.2.3 Machine Learning

In contrast to classic *logic-based* artificial intelligence, an alternative tactic could be implementing an agent that adapts from experience rather than being explicitly programmed how to react under specific circumstances. This is called *machine learning*.

Machine learning can be categorised into *three types* depending on how an algorithm is designed to learn. The first is *supervised learning* where the algorithm is provided with a *training set* which contains correct outputs for all the inputs given. Using this, it will be tweaked to react correctly for any possible input. Examples that used *supervised learning* include *artificial neural networks* and *support vector machines*. The next is un*supervised learning* which is not supplied with a *training set*. Instead, it will attempt to group the common elements of the inputs given like in the clustering algorithm *k-means* [34].

Neither of these learning types is usually implemented in game-playing AI as *supervised learning* requires the algorithm to know all outputs for every specific move and *unsupervised learning* will not use the feedback given from past move outputs. This being said, *training set* data can however be explicitly provided by a human expert (such as '*Neurogammon*' which used *neural networks* for evaluating backgammon positions) but the algorithm produced will never be able to play the game at a greater level than the expert [28].

The final type of *machine learning* addresses these two issues. This type is called *reinforcement learning*. Using this method, the algorithm is informed when outputs are wrong however isn't given any information regarding how to correct them. Because of this, the algorithm must try different possibilities until it can find how to get the correct outputs for specific inputs given [34]. Therefore, *reinforcement learning* is mostly used in game-playing AI as correct moves do not need to be given prior for the agent to gain feedback on whether the moves it is choosing are helping it win. Below are some commonly used reinforcement techniques used in games.

Q-Learning and SARSA

Using any given *Markov Decision Process (MDP)*, policy learning algorithms such as *Q-Learning* and *SARSA (State-Action-Reward-State-Action)* can be used to learn *optimal*

policies. *Policies* are used to decide what the next best action is to take in a specific state, thus in games it is used to pick the next best move. However, both algorithms differ with the strategies they both produce. *Q-Learning* always attempts to follow the shortest path in order to gain maximum *reward*, thus is more prone to take risks, as the shorter path may lead to a negative outcome. *SARSA* will avoid taking risks to learn the *optimal policy*, which means it will always try and act cautiously but may take longer to learn [34]

TD-Learning

Combining both *Dynamic Programming* and *Monte Carlo* methods, *Temporal Difference (TD) Learning* can be used to predict the next best action to take in a given scenario. In this procedure, a full model of the environment is not required and updates are made to the algorithm at every new state it encounters. Thus, TD-Learning adapts only from new experiences in partially unknown domains [35]. This is an appropriate method to be used for *stochastic games* involving *imperfect information*, as full environments do not need to be known and the chance elements can rely heavily on the next state predictions it makes.

2.2.4 General Game Playing

More recently, studies have started to focus on developing systems that have the capability of playing more than one game using only the *descriptions* (rules and setup) provided. This is called *General Game Playing (GGP)*. Using well known AI methods such as *Knowledge Representation*, *Reasoning* and *Machine Learning,* a *GGP* program will incorporate these to interpret game descriptions to play at a reasonable level instead of relying on algorithms created prior for a particular game [36].

GGP programs must also be able to play a range of games that differ in both complexity and characteristics e.g. *perfect/imperfect information*, *deterministic/stochastic* moves etc. There have already been successful *GGP* programs implemented that can play the arcade games '*Space Invaders'*, '*Lunar Lander'* and '*Frogger'* using only the descriptions provided for each game [37].

2.3 The Game of 'Jaipur'

The game which will be explored in this project is called '*Jaipur'*. This is a *strategy-based* card game involving two players. To date, there has been no research dedicated to finding any winning strategies for the game most likely because its characteristics differ somewhat from games traditionally studied. It can also be argued that due to its fairly unknown reputation when compared to other popular card games such as *Poker*, *Rummy* and *Blackjack*, it has received a lot less attention for the means of academic study.

I feel the overall complexity of the game is at an appropriate level for the time available. This being said, the game is not simple and requires a lot of strategic thought when making decisions meaning the results I should obtain will be useful in similar future studies. Also, as research into the game is unprecedented, my techniques, findings and conclusions will not be affected by any other studies about the game itself.

2.3.1 Setup

The game consists of four main components: *goods cards*, *camel cards*, *goods tokens* and *bonus tokens*. The goods cards come in 6 types, each representing a different commodity. The commodities are: *diamonds*, *gold*, *silver*, *cloth*, *spices* and *leather*. All cards of the same commodity type are identical, but the value of a card (when it is 'sold') depends both on the commodity type and on the number of cards of that type that have already been sold (thus, the value decreases as more cards are sold). Bonus tokens can either represent a three, four or five multiplier bonus, with the average token value increasing respectively. Camel cards have no variation, but do have a single corresponding bonus token worth 5 points.

Before any game begins, each goods token must be sorted in descending order (largest on top and smallest on the bottom) in each commodity type. The bonus tokens are shuffled in their specific multiplier groups. Three camel cards are then placed face up between both players which form the market cards of each round (this also can be referred to the 'market place'). The remaining cards in the deck are shuffled. Two cards are then also placed face up in the market place, and an additional five cards are distributed between each player. Both players must then extract the camel cards from their hand and place them in a stack face up in front of them, this is called the player's 'camel herd'. The remaining cards form each player's goods hand which are not to be shown to either opponent.

Figure 2.4 – A typical setup of the game 'Jaipur', taken from an online version of the game [38]

2.3.2 Rules

The main focus of '*Jaipur'* involves *taking*, *selling* and *trading* cards to try and become richer than your rival trader (the opponent) after every round. The first player to win two rounds is the winner.

The goods cards represents what each player currently has in their disposal, what goods are currently in the market place and what goods have already been sold (burnt cards). The camel cards are mainly used for trades in the market place when more than one goods card is of interest to the player. The goods tokens represent how many points a player has earned when selling one or more cards of the same type. Bonus tokens can be gained when selling three, four, or five of the same card in one turn or having more camels at the end of a round, both resulting in extra points.

A player can choose one of four actions in their turn: *'take all camel cards'*, *'take one goods card'*, *'sell cards'* or *'trade cards'*. If a player wishes to take cards from the market place, they can either take one goods card from the market and replace it with one from the deck, they can take all of the available camels and replace them all with cards from the deck, or trade two or more cards (camels or goods cards in the player's hand) with goods cards in the market. It should also be mentioned that a player can only have up to 7 goods cards in their deck at any one given time, any amount of camel cards is allowed.

However if a player wants to sell cards they can select one or more goods cards (of the same type) from their deck and sell them in exchange for the correct number of goods tokens (and bonus tokens if applicable). If a player tries to sell more cards than there are available tokens, they will instead only take all of the available tokens of that specific goods type, this does not however stop the player achieving a bonus token if three or more cards are sold. If the player wishes to sell expensive goods (*diamonds*, *gold* or *silver*), two or more must be selected unlike the other cheaper goods (*leather*, *fabric* or *spices*).

A round ends if there are no more cards left within the deck or if there are no more tokens left for three good types. After each round the player with the most camels achieves the bonus camel token worth five points, if both players have the same amount of camels, neither gain the token. Once this is addressed, all the tokens a player has gained are totalled and the player with the most points wins the round. If a draw still occurs, the player with the most bonus tokens wins, and then with the most goods cards if there is still a draw. Two won rounds for a player results in an overall game win.

It should also be noted that in every round each player is permitted to know the exact quantity of goods cards the opposing player currently has and whether they possess camel cards or not.

2.3.3 Classification

Below, using the game categories previously defined in *Section 2.1.3, the* game '*Jaipur*' has the following characteristics:

- *Zero-Sum*: If we are to treat each game victory as the ultimate payoff for each play (+1 for a win and -1 for a defeat), it is not possible for both players to win or lose.
- *Two Player*: Playing the game with a single player or more than two is not possible.
- *Imperfect Information*: Each player cannot see their opponent's goods cards nor can they explicitly know how many camel cards they currently have.
- *Stochastic*: If a market card is taken and not replaced, one is taken from the shuffled deck and placed in the market place.
- *Non-Cooperative*: As the game contains two players, cooperation is not possible.
- *Sequential*: Moves have to be taken in a consecutive manner, and each player must wait for their turn in order to perform a move.

As *'Jaipur'* involves *imperfect information* and is defined as *stochastic*, it deviates from other games which have been studied extensively in the past, and therefore is a good choice of game to study for the overall purpose of this project.

2.3.4 Approach

To create the artificial player for the game, I have decided to use the *expectiminimax* algorithm which will incorporate *heuristic state evaluation* functions for any *pre-selected depth* of the *game tree*. Using this, the AI will be able to decide what the next best move is in order to receive the *optimal payoff* in any given state. I have noticed through playing the game multiple times, “almost” *perfect information* can be achieved by studying the previous game states and every move available to the player does not necessarily have to involve chance.

Although there is always some uncertainty at the beginning of the game, since new cards can only be obtained by drawing from the market place, it is nearly always possible to obtain *complete knowledge* of the cards the opponent holds within a few moves of the game start. It is because of this, I have decided to implement *logic-based* methods used more in traditional games opposed to using an alternative *machine learning* approach. I have also avoided using *General Game Playing* methods as only one game for this project is being studied.

As previously defined in *Section 2.3.3*, a player can choose between a total of four actions depending on the type of move they wish to pursue. Each action has a different number of possible moves depending on its type and the cards available within the player’s current hand and the current market place. As illustrated in the table below, in any given turn, a player will only be able to choose between a maximum of 6 *stochastic* moves out of a potential 38. Therefore, it can be argued that even though the game is classified as *stochastic*, a large amount of moves available in any given turn can be *deterministic*.

Action	Maximum Number of Moves Available	Move Type
Take all camel cards	1	Stochastic
Take one goods card	5	Stochastic
Trade cards	26	Deterministic
Sell cards	5	Deterministic

However, it is worth noting that what is directly obtained by the player in terms of cards or points is always *determined*. The *stochastic* element only arises when cards are taken from the market and replaced from the main deck (rather than from a player’s hand in the ‘trade cards’ action). Thus, the *randomness* only affects the possibilities of the next state.

In addition to this, the *imperfect information* within '*Jaipur'* can be completely eradicated if the player is able to correctly evaluate the opposition's current hand. This can be achieved by taking into account the opposing player's *past moves* and by using the *information currently known* about the opposition. This is only possible however due to the fact a player cannot directly pick up a card from the shuffled deck unless it is within the initial setup of the game where 5 are distributed randomly to both.

Because of these two factors, the *expectiminimax* algorithm only has to incorporate chance nodes for *stochastic* moves but can operate like the original *minimax* algorithm otherwise. Thus, *strategies* will only use a *mixed* approach if a move it is handling involves *uncertainty*. Alternatively, if the move chosen is *deterministic*, a *strategy* should only use a *pure* approach. The *depth limit* for each *game tree* will ultimately be dictated by the *knowledge* the current player has regarding the opposing player's hand. This information is vital to predict how the other player is going to react and perform in future moves; therefore fluctuation in *game tree depth* must be considered.

It has also been decided that each round will be categorised as a *single play* of each *game* (rather than the `best of three' specified in the official rules) due to every round being unique and independent from the last. Therefore the *ultimate goal* for the artificial player will be to win each round it is currently playing, thus making the *overall payoff* for each *play* the *round score* and not the overall *game victory/defeat*. If this approach is taken for every round, winning the overall game can be achieved as a consequence.

Two types of *strategy* will be created for the artificial player to use within any single *play*. The first strategy type will use a *basic approach* that will rely on both *random* and *greedy* decision making. The second will use the more suitable *expectiminimax approach* with *heuristic state evaluation*. This will differ in *depth limit* depending on the specific strategy chosen. The purpose of this will be to test the strategy types against one another in order to evaluate the strength of the *expectiminimax algorithm* implemented. Before this however, the most effective parameters within the *heuristic state evaluation* will be found through playing multiple instances of the game using two AI players, each using a different strategy type.

After this, the best strategy found will be used to compete against three separate human players in a multiple number of *plays*. The test results should be able to primarily decide whether the main aim of the project has been met or not.

Chapter 3 – Game Implementation

To be able to implement an artificial player, I need to first produce a platform in which the game can be played between two human players. This will consist of integrating all the major components and aspects of the main *setup* and *rules* discussed in *Sections 2.3.1 and 2.3.2*. To do this, I will first justify the language which I have chosen to write the software in. This will then be followed by a summary of all the classes I have designed which contain the core elements found within the game and a brief overview of some of the key attributes I have included. To finish, I will discuss the changes I have made in regards to the final interface chosen for the main solution.

3.1 Language

Through playing the game multiple times, I have found that typically each player will have 20-25 turns per round. This means the total number of turns in each round can approximately be up to 50. Given that in each turn a single player can have a total of 37 possible choices, the *worst-case search space* within a single round can potentially be 50^{37} ($\approx 7.276 \cdot 10^{62}$). Because of this, I needed to select a programming language that was capable of handling large amounts of data within an efficient amount of time.

For this problem, I chose to select *Java* as the main language to implement my solution for three of its main properties: *efficient speed*, *automatic memory management* and *rich object-oriented design*. Unlike lower-level languages such as *C* and *C++, Java* offers *garbage collection* which automatically removes old unused memory when processing large amounts of data (such as calculating all possible game states). This being said, the same procedure can be done manually in *C* and *C++* however is prone to cause many unnecessary errors [39] which was very undesirable given the project timescale.

Even though *Java* has been proven to be faster than languages with simpler syntaxes such as *Python* and *Perl*, typically *Java* is slightly slower than *C* and *C++.* However, as neither *C* nor *C++* offer *garbage collection*, *Java* was still the stronger choice.

I decided that having an object-oriented solution was to be the most appropriate approach for representing the game. This is because I knew the core components could each be separately represented as collections of objects. Each object would be able to have relative methods associated to them which could abide by the games rules thus making illegal moves impossible to perform. Also making additions and amendments to the foundations of

the main game and AI code could be easily accomplished. As *Java* insists on using an object- oriented approach, it seemed wise to select this language.

3.2 Class Structure

To separate the main game into classes, I first established the main psychical components. This consisted of all the cards and tokens used throughout each round, thus three classes were established: *MainDeck*, *GoodsTokens* and *BonusTokens*. The *MainDeck* class represents all 55 cards that can be placed in the market and handled/sold by both players. Both token classes (*GoodsTokens* and *BonusTokens)* alternatively represent the goods and bonus tokens that can be obtained after any sale is made by either player. All of these classes contain methods that adjust in accordance to player moves and determine when a round has ended (e.g. when there are no more cards left within the current deck to distribute).

After this, a *Player* class was created to handle every current hand and point score of each participant within a round. This class mainly adds/removes cards from the current player's hand given their move choice and adds points to their score if a sale has occurred. This was then extended into the two subclasses *HumanPlayer* and *ComputerPlayer* which determines whether the player is human or synthetic. The *ComputerPlayer* class will be expanded later once the AI implementation has begun.

These classes were then all collectively used to create an instance of each round in a new class named *JaipurRound*. This was primarily designed to introduce the initial setup of a specific round, explicit card domains (the market place and previously sold cards) and the rules used within the general gameplay including legal move choices each player can make.

Three instances of the *JaipurRound* class were established in a new class representing the full game named *JaipurGame*. In this class, an instance of each round can be created and played using two players. Round wins are determined and totalled, player turns are established and user interaction (e.g. selecting the next move to take) is controlled through a text-based interface contained within a separate class named *TextInterface*. This interface is predominantly used to convey all of the relevant information a user needs in order to successfully play the game e.g. the current game table, the current player name and score, and the visible information known about the opposing player (such as goods cards quantity). Also, this is where a user will be able to input commands such as move choices and listing both sold cards and previous moves already executed by either player.

Each game can be played through a class that creates a single instance of *JaipurGame*. This class is called *JaipurMain*. Later, once the artificial player has been created, I will use this class to test the parameters used within the *heuristic state evaluation* and to compare the various *strategies* I will eventually implement for the AI.

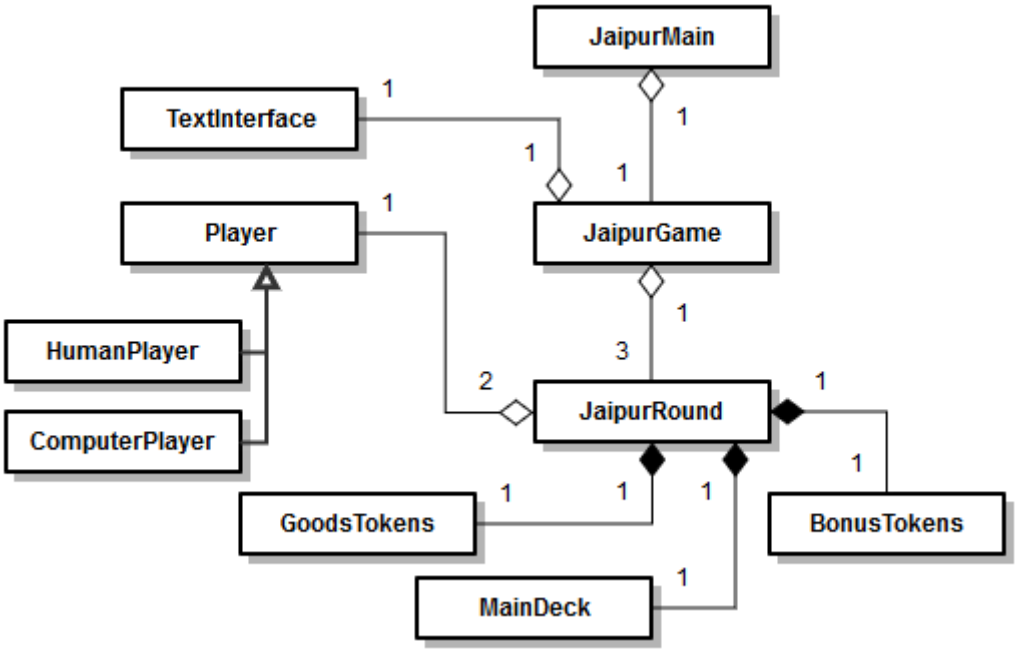


Figure 3.1 – A UML diagram displaying the basic composition and relationships between each class

3.3 Key Class Attributes

In order to understand how I will design the artificial player for the game, I first need to provide a brief overview of some of the key attributes I have included within the classes. Each attribute I will discuss contains necessary information needed for the AI *strategies* I will later implement.

All of the *cards* and *tokens* within each round are handled within separate containers using the *ArrayList* class. Each individual *card type* is represented by a unique *string* consisting of the first character or first three characters of each card name, depending on whether two card names share the same first letter. Therefore, each card type can be represented by one of the following strings: "*D*", "*G*", "*Sil*", "*Clo*", "*Spi*", "*L*" and "*Cam*". Each *token* alternatively is assigned to a single *integer* value.

To handle every *card* and *token* within each round, domains have been produced through using particular attributes within specific classes. These attributes use *ArrayList* data structures, each one consisting of either *strings* for card types or *integers* for token values.

A single card can be handled between six main domains once it is has been taken from the *MainDeck* class. These include '*current_goods_hand'* and '*current_camels_hand'* in two instances of the *Player* class, and '*current_market'* and '*sold_cards'* in one instance of the *JaipurRound* class. Cards can only be reallocated to other domains if they are used in a move performed by a player. This has been illustrated in *Figure 3.2* below.

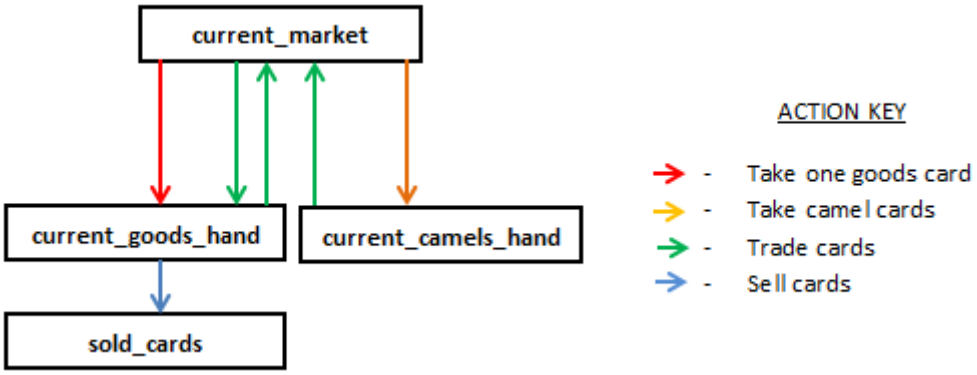


Figure 3.2 - A diagram displaying which domain each card is transferred to after a specific action type is performed.

Each token type is represented by eight domains. These include '*diamond_tokens'*, '*gold_tokens'*, '*silver_tokens'*, '*cloth_tokens'*, '*spice_tokens'* and '*leather_tokens'* in one instance of the *GoodsTokens* class, and '*five_multiplier_tokens'*, '*four_multiplier_tokens'* and '*three_multiplier_tokens'* in one instance of the *BonusTokens* class. Goods tokens are initialised in descending order, whilst bonus tokens are initialised randomly. If a sale move is chosen by a player, the suitable amount of goods tokens are removed from the appropriate container in the *GoodsTokens* class. If a bonus token is permitted, one is also removed from the appropriate container in the *BonusTokens* class. The integer values extracted are then added to the player's overall round score.

Every move performed by a player is represented in an *ArrayList* containing strings. This is named '*next_move*'. The first element always contains the action type of the move chosen. Additional elements in each array represent the cards involved. Below are examples of the '*next_move'* container used for each action type:

- `["take_camels", "Cam", "Cam", "Cam"]`
- `["take_one_good", "G"]`
- `["trade_cards", "L", "Cam", "Cam", "--", "Sil", "Spi", "D"]`
- `["sell_cards", "Clo", "Clo"]`

Every instance of '*next_move'* is added to an attribute within the *JaipurRound* class named '*previous_moves'*. This is an *ArrayList* that contains all past *ArrayList* instances of the '*next_move'* container, thus saving all previous moves chosen by both players.

3.4 Interface

As previously mentioned, users can interact with the game using a clear *text-based interface* which dispalys all game state information visible to the current player (e.g. the remaining goods tokens, the cards in the market place and the current player's hand etc). This was originally intended only to be part of the basic solution, however after a discussion with my supervisor it was decided that it would remain. Initially, a graphical user interface was planned to be written using a *Java API* such as *Swing* or *AWT*, however due to time constraints, unexpected bugs and its overall importance to the project aim, the idea had to be dropped. In retrospect, I feel it was the correct judgement to make as the final solution created should focus on the efficiency and performance of the artificial player and the interface design has no impact on this factor.

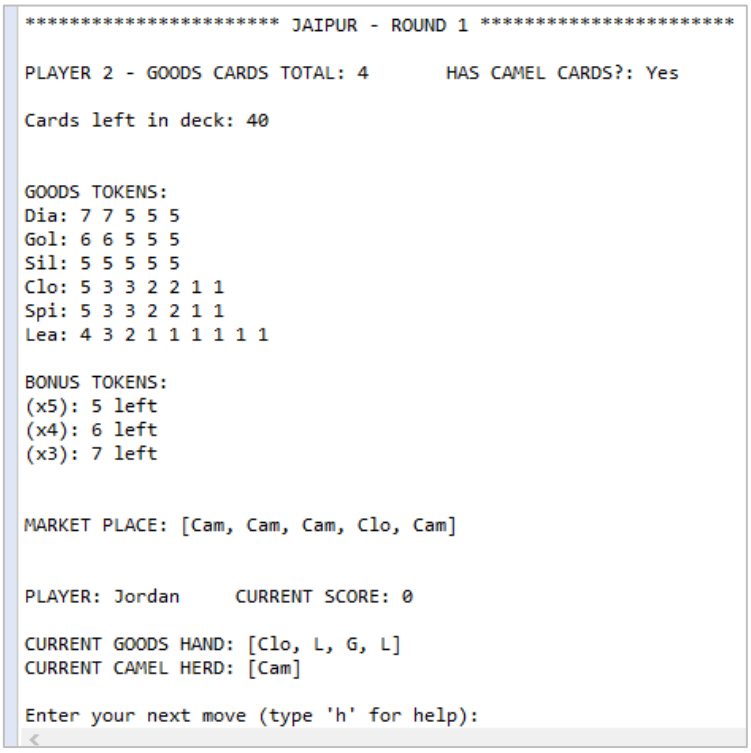


Figure 3.3 – A screenshot of the text-based user interface

Chapter 4 – Constructing an Artificial Player

In this chapter, I will provide a general outline of all of the stages I conducted to create the final AI solution. Through using the *ComputerPlayer* class, I separated the task by assembling multiple methods which have been collectively used to build both *basic* and *advanced game strategies*. I will first provide details of how I have represented and modelled each *game state*. After this, the steps taken to implement each *strategy* will be explained and the details of how each one operates will be provided. Please note, any pseudocode referred to in *Sections 4.2* and *4.3* can be found in *Appendix C*.

4.1 Game State Representation

Before any intelligent decision making could be made by the AI, I needed to first provide it with all the necessary information about the current state of the game. Using specific attributes from other classes (discussed in *Section 3.3*), I tried to simplify all visible information numerically using one-dimensional arrays. This approach would allow the program to represent each state in a more compact way. This idea would later become critical as calculating future states of the game can be exponentially large in number.

At any given state, a player is able to identify all the cards within their current hand, the market place and the sold pile, all the goods tokens currently available and all the previous moves performed by both players. As all of the token containers (such as '*diamond_tokens'*, '*gold_tokens'*, '*silver_tokens'* etc.) only hold the *integer* value for each available token, no adjustments needed to be made to them. However, as each card domain uses *strings* to represent each card, an alternative representation was required.

To overcome this, a method was developed which takes a specified domain, and totals the quantity of each card type into an array. For clarity, the quantities of each domain are always presented in the following order: *diamond total*, *gold total, silver total, cloth total, spice total, leather total* and *camel total*. As both goods cards and camel cards were included in this array, combining both the domains '*current_goods_cards'* and '*current_camels_cards'* from the *Player* class was necessary to find the quantities of a current player hand.

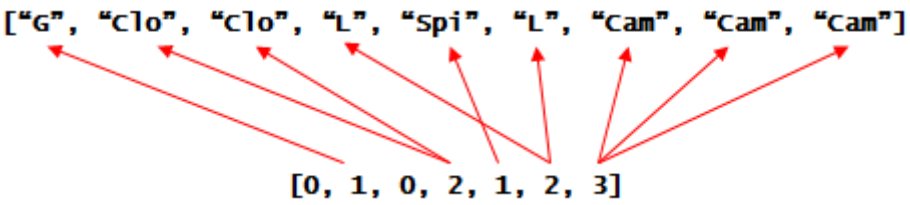


Figure 4.1 - An example of a card domain converted into a one-dimensional array of card quantities.

4.2 Basic Strategies

Before I focussed on implementing the *expectiminimax* algorithm, I wanted to first create *basic strategies* that only took *naïve approaches* when choosing what move to perform next. These were intended to imitate an *inexperienced player* that only considered their current/next hand whilst ignoring the opposing player, the cards within the current market place and any potential drawbacks in the near future. The purpose of this was to evaluate if such a *strategy type* could successfully be used to win a game, especially when playing against another *strategy* that incorporated *expectiminimax*.

In order for the artificial player to be able to make any decisions at all, it would first need to be provided with a set of *all possible moves* to pick from within a specific state. All moves found needed to be valid and therefore were required to be legal in accordance to the rules of '*Jaipur*'. This was achieved by analysing the *current market cards* and the *current player's hand*. Eventually, this would be used by *every* strategy created.

4.2.1 Random Decision Making

The simplest way for a human player to make a choice is by thinking *irrationally*. I realised the same concept could be applied for the AI when it is choosing what move to perform next. If a move is chosen at *random*, the AI is not taking into account any other factors of the game's current state. Therefore, if this approach was adopted for every single decision made within a particular round, no measures will have been taken to ensure that the player gains any points through *selling goods cards*.

This does not however mean that using a *random approach* is entirely useless. Whilst only selecting random moves does not ensure round *victory*, it does not necessarily ensure *defeat* either. Also, using a *random decision process* may be convenient to use when there are no more options left to select when employing another *non-random* approach. All things

considered, I decided that only relying on chance to make decisions seemed a very unwise tactic to take, especially when the opponent shows traits of *rational behaviour*.

4.2.2 Greedy Decision Making

A way to improve upon selecting moves only at *random* is by also considering the move which guarantees the *highest immediate payoff*. This is an example of a *greedy choice*. If applied to '*Jaipur'*, using this approach would not consider any potential *future rewards* (e.g. trading cards for a more valuable hand) but would always select the move that provides the player with the most points given the goods tokens still available. Thus, if a sale action was possible, one would always be selected when using a *greedy strategy*.

However, I realised that if a sale action could not be made within a given state, the AI would still require a way of deciding what move to choose next. To overcome this issue I developed two separate protocols, either of which could be applied in such a scenario. The first just selected a move at *random*. The second took a more logical approach and aimed to maximise the value of the hand obtained in the next state using the remaining goods tokens available (see *Pseudocode C.1* for details). It should be noted that this approach does not value *camel cards*, and therefore will not purposely aim to gain the *bonus camel token* gained by one of the players at the end of each round.

4.2.3 Summary

Through using both the *random* and *greedy decision processes* discussed, I have created three separate *basic strategies* that can each be used in a single play of '*Jaipur'*. Below I have summarised how each strategy selects its next move within any state of the game.

Basic Strategy 1

Every *move* is chosen completely at *random*.

Basic Strategy 2

If there are one or more *moves* available where a *sale* is possible, the one which attains the *highest amount of points* is chosen; otherwise, a *random* move is chosen.

Basic Strategy 3

If there are one or more *moves* available where a *sale* is possible, the one which attains the *highest amount of points* is chosen; otherwise, the *move* which provides the player with the *most valuable hand* (using *Pseudocode C.1*) is chosen.

4.3 Advanced Strategies

In the next stage, I concentrated my attention to focus primarily on recreating the *expectiminimax* algorithm that would successfully integrate all of the state components '*Jaipur'* had to offer. To do this, I would need to calculate the *stochastic information* presented in the *market place* and then abolish all *imperfect information* within each *play* by successfully *evaluating the opponent's hand*. This was followed by creating a *heuristic evaluation function* which would be applied to approximate the overall payoff of each possible move to select from once a *specific depth limit* of a *game tree* was reached. Finally, these concepts were assembled to produce the *final algorithm*.

Three *advanced strategies* were then devised using *expectiminimax*. The main goal for these approaches was to consider all other factors of each state, which all the previous *basic strategies* had disregarded.

4.3.1 Estimating Future Market Cards

To be able to represent *chance nodes* within each game tree, I had to find a suitable way of conveying all of the *stochastic information* found within future states. As previously mentioned, the only *uncertainty* offered within the game are the cards that are taken from the *shuffled deck* and added to the *market place* after specific moves are performed (*'take all camel cards'* and *'take one goods card').* To simplify the problem, I needed to find the *probabilities* of each specific *card type* being taken from the *main deck* using the information available to the player. Thus, every time a new card was to be added to the market after a *stochastic move*, an estimation of the next market card quantities could be calculated using the *probabilities* of all remaining cards not currently visible to the player.

Taking this approach would mean that *chance nodes* would not need to be considered within the main body of the *expectiminimax* algorithm as the *next market card quantities* would always factor any *ambiguities* presented by the *shuffled deck*. Because of this, *deterministic* moves would avoid any additional calculations involving probability and therefore the algorithm would potentially be able to increase its general *time efficiency*.

To achieve this, I first found the quantities of each card type outside of the main deck known to the player (such as the player's current hand, the cards in the current market place and cards that had been sold in previous states). The maximum possible quantity of each card type was then subtracted by these values to find all remaining quantities of each card type still within the main deck. Once these were acquired, they were used to find the probabilities of the next card taken from the deck being of a certain type. This was done by taking the remaining value of each card type and dividing it by the total number of cards remaining (see *Pseudocode C.2* for details).

I then chose to adopt this method to estimate what the market card quantities would be after a *stochastic* move was chosen. Using determined market place quantities already found within the next state, each remaining card probability for each card type would be added. Depending on how many cards were taken within each move would ultimately dictate how many times the remaining card probabilities were to be added (see *Pseudocode C.3* for details). For example, if a '*take all camels*' move was selected and there were three camels available, the remaining card probabilities would be added three times to the determined card quantities of the next market. Using this method would allow the AI to calculate an estimation of the next market within any given state.

It should be noted that the probabilistic market quantities calculated using this method are somewhat different to the actual probabilities of possible subsequent market states. In general, when one card is taken and replaced from the deck, there will be up to seven possible next market states depending on what card is drawn. Each will have a certain probability of occurring but will always have an integer number of each card type. This would give a large branching factor. For example, if n camel cards were taken from the market place, the branching factor would be n^7. Because of this, the method I have implemented aims to provide a good approximation for the actual probabilities instead to cut down on search space.

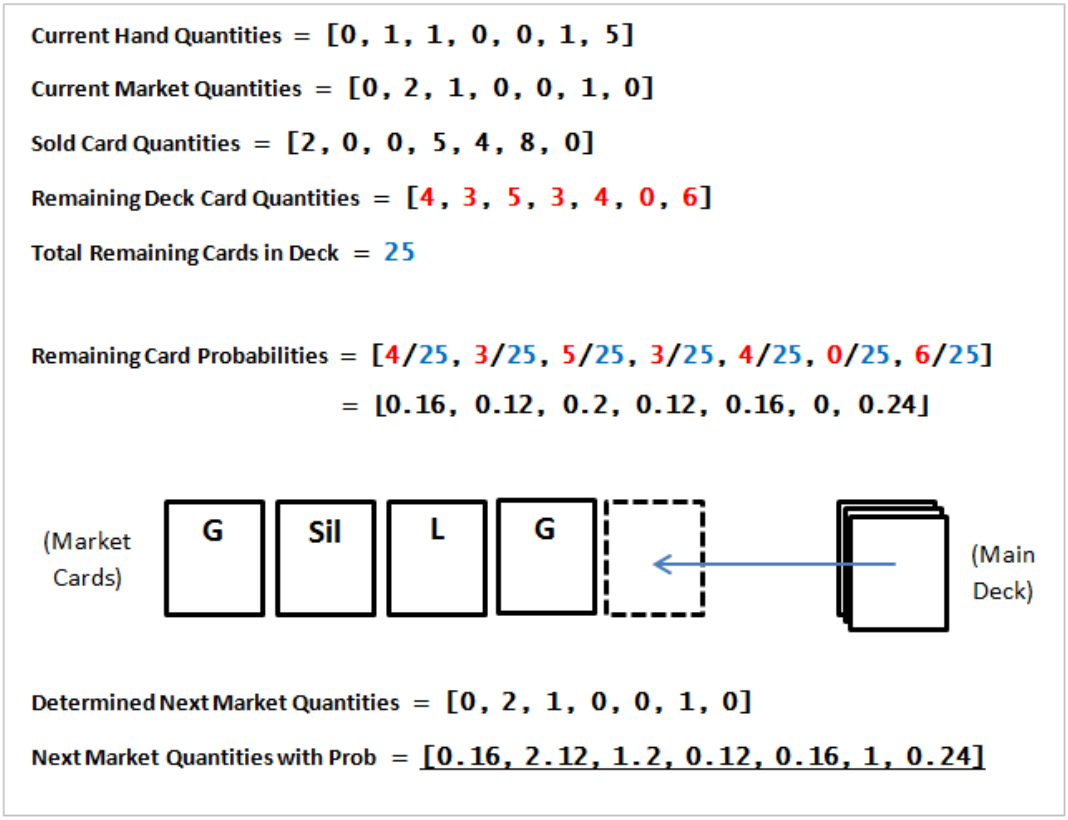


Figure 4.2 - An example of the calculations involved when estimating the next market quantities for one undetermined new card

4.3.2 Opposition Hand Prediction

In order for the *expectiminimax* algorithm to be able to operate correctly, all information about the current state of the game is required to be known by the AI. This was an issue as the rules of '*Jaipur*' do not allow any player to see their opponent's cards once a round has officially begun. Because of this, I needed to develop a method that would be able to successfully evaluate the quantities of each card type within the opponent's current hand.

Even though a player cannot explicitly know the contents of their opponent's current hand, they are permitted to know the total number of goods card the opponent possesses, but not how many of each type. From this, I was able to find the opponent's initial camel total by subtracting the known amount of goods cards from the amount of cards always given to each player at the start of each round. The initial quantities for each type of good card would however remain unknown. Therefore at the start of each round two properties of the opponent's hand could be established. These were the initial quantity of camel cards and the initial number of goods cards unknown. Using this information, an array would then be

generated presenting the initial known cards possessed by the opponent (e.g. [0, 0, 0, 0, 0, 0, 3]). To begin with, the array values will always be 0 for each good type, since only the number of camel cards is known.

After this, I discovered that through analysing all of the moves performed by the opponent in previous states, the array could be updated to identify the known quantities of each card type. As each possible move in '*Jaipur*' consists of either adding or removing cards from a player's hand, the quantity of a specific card type would be amended using just this information alone. Thus, the remaining number of unknown goods cards would decrease every time there was a specific goods card taken from the opponent's hand which was not previously known to be there. When the number of known cards becomes equal to the total number of cards held, all cards are known and this will then remain the case for the rest of the game. Usually this will happen within only a few moves.

It is also worth mentioning that depending on the type of move selected, the number of cards either added or removed from each hand would vary. To illustrate this, I have summarised how each action type affects the known quantities of the opponent's hand:

- "*take_camels*" – increases the camel card quantity by however many camel cards are taken from the market place.
- "*take_one_good*" – increases a specific goods card quantity by 1.
- "*trade_cards*" – increases specific goods cards quantities and decreases specific goods and/or camel card quantities by the number of cards being exchanged with the market place.
- "*sell_cards*" – decreases a specific goods card quantity by the number specified within the sale.

As an example, I have provided a table that demonstrates how the quantities of each card type within the opponent's hand are evaluated in every round.

Turn Number	**Total Number of Goods Cards**	**Known Card Quantities**	**Unknown Card Total**	**Move**
1	3	[0, 0, 0, 0, 0, 0, 2]	3	["take_one_good", "G"]
2	4	[0, 1, 0, 0, 0, 0, 2]	3	["sell_cards", "Clo"]
3	3	[0, 1, 0, 0, 0, 0, 2]	2	["trade_cards", "G", "L", "—", "D", "D"]

4	3	[2, 0, 0, 0, 0, 0, 2]	1	[“take_camels”, “Cam”, “Cam”]
5	3	[2, 0, 0, 0, 0, 0, 4]	1	[“trade_cards”, “D”, “Sil”, “—“, “Spi”, “Clo”]
6	3	[1, 0, 0, 1, 1, 0, 4]	0	-

4.3.3 Heuristic State Evaluation Function

The *heuristic function* was initially designed with the intention of being able to calculate the value of a specific state. This was to be achieved using only the information known about the state and the previous move used to get there. As the main objective of each round is to gain more points than the opponent, I decided to focus the heuristic towards achieving this goal. Because of this, the points gained in the previous move (including any bonus averages obtained) and the value of the hand in the current state both needed to be considered. I also needed to take into account the value of the estimated market to minimise the opponent’s chances of using it for potential future profit. Thus, the following function was developed:

```
f(h) = points_gained + hand_value – market_value
```

The next challenge involved separately finding the values of both the hand and the estimated market within the given state. To accomplish this, I first reflected on the algorithm I used within *Basic Strategy 3*. This evaluated all potential next hands based only on the goods tokens remaining (see *Pseudocode C.1*). The method itself was a good way of evaluating the absolute value of each goods card within a given domain, however it did not take into consideration other components of the state which could be critical for the future success of the player.

Such components included any camel cards present and any bonus tokens remaining. Because neither of these were factored, there would be no effort to try to acquire the camel token (worth 5 points) and situations showing great promise in profit through gaining bonuses alone could be completely overlooked. An example of this would be if a player possessed 5 leather cards and no leather tokens remained. In this scenario, the algorithm would evaluate that their hand was worthless. This typically would be an incorrect judgement as an average of 9 points could still be attained in a sale through acquiring a single bonus token.

The algorithm would also assume that any quantity of each goods type within a given domain would always be sellable if greater or equal to 1. This assumption however is untrue for when the quantities of the *diamond*, *gold* and *silver* cards are lower than 2. In addition to this, if the same algorithm was applied to a domain that included additional card probabilities (such as the quantities found in a market after a *stochastic* move is performed), each probability would be disregarded and each quantity would essentially be truncated.

In order to address these issues, I developed an alternative algorithm that would incorporate *adjustable heuristic parameters*. This method would first find the value of each goods type by using the quantities present within the domain provided and the corresponding goods tokens that remained. Any additional probability within each quantity would also be included in this calculation. If the quantity of a goods card type was greater than or equal to 3, a bonus average would then be added to the value of that specific goods type.

If the quantity of a card type permitted a sale action, the value calculated for that card type would be multiplied by the parameter k_1. However, if the quantity of a card type did not permit a sale action, the value calculated for that card type would instead be multiplied by the parameter k_2. After this, the new value for each goods type would then be summated to find the overall value for all goods cards. The value for each camel card would then be found by multiplying the camel total by the parameter k_3. Thus, these two values were finally added to produce the overall value of the given card domain (see *Pseudocode C.4* for details).

A simplified version of the algorithm can be presented by the following *heuristic formula*:

$$\begin{aligned} \texttt{card\_domain\_value} = \; & k_1 * (\texttt{sellable\_card\_values}) + \\ & k_2 * (\texttt{unsellable\_card\_values}) + \\ & k_3 * (\texttt{camel total}) \end{aligned}$$

Using this, I was then able to determine the values of both the hand and the estimated market within a given state. Thus, the heuristic value of each potential state could now be found (see *Pseudocode C.5* for details).

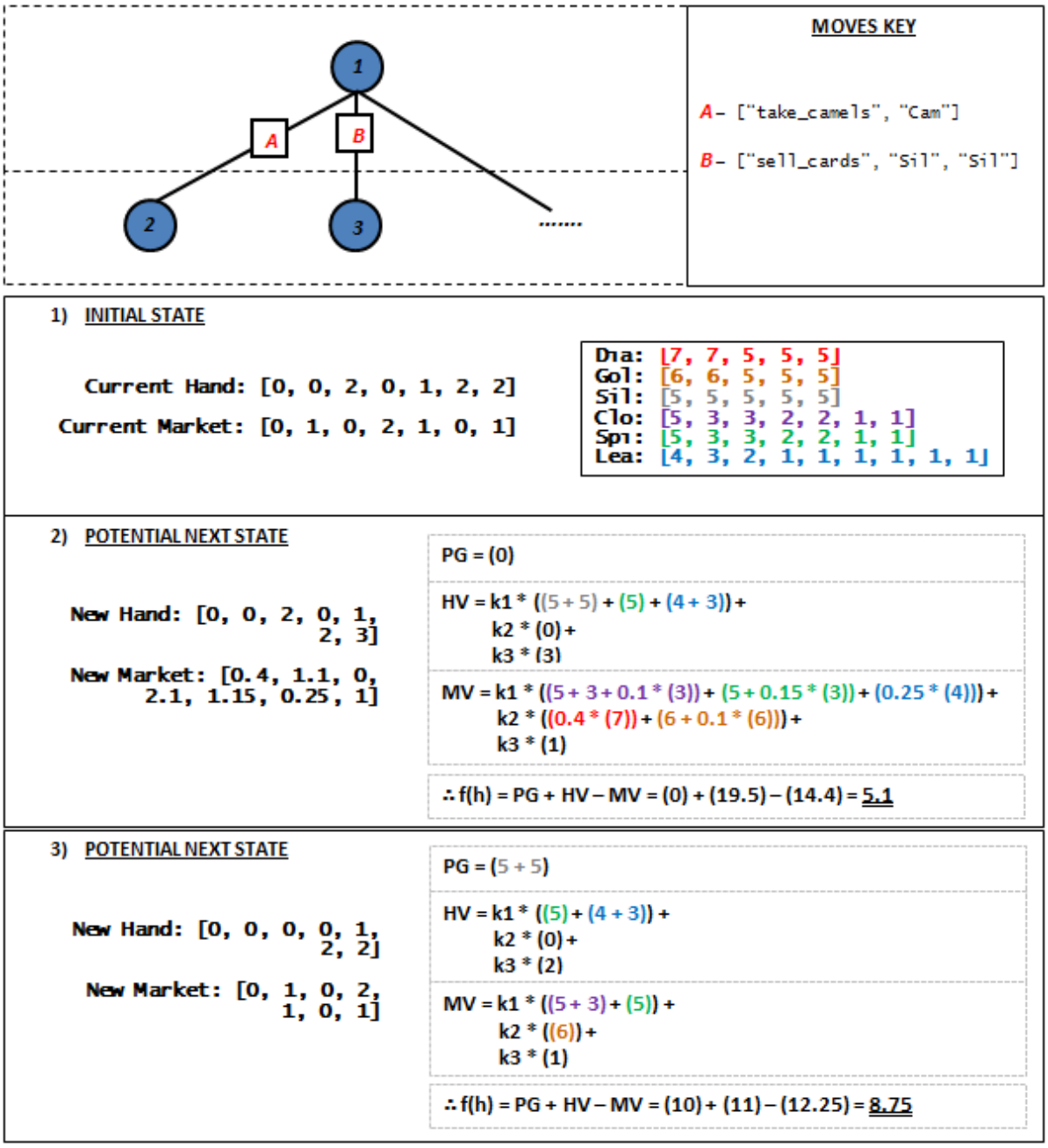


Figure 4.3 - An example of the calculations involved when evaluating two potential next states using the heuristic parameters $k_1 = 0.75$, $k_2 = 0.25$ and $k_3 = 1$.

4.3.4 Expectiminimax

Using all of the components I had previously established in *Sections 4.3.1, 4.3.2* and *4.3.3*, I finally had all the tools necessary to fully implement the *expectiminimax* algorithm. As *chance nodes* were now not being included within the main body of the algorithm, I was able to use the pseudocode from the previous research I had conducted earlier (see *Figure 2.3* in *Section 2.2.2*) for the main inspiration towards its overall design. However, slight adjustments would need to be made in order for the cutoff testing to be possible. The testing itself would use both the *heuristic evaluation function* and a pre-specified *depth limit*.

I first considered all of the features within each current state needed to represent all possible future states. This included the information known about both players (hand quantities and current scores), the information known about all other shared elements of the game (current available goods tokens, remaining card number, previously sold cards and the remaining card probabilities) and all possible moves the current player can take.

Through recursion, the algorithm would then use a *depth first search* to find all possible future states within the game tree, whilst updating information about each one (e.g. increasing each player's score, removing sold goods tokens, adding sold cards and so on). If a depth limit was ever reached, the move that provides the highest value based on the evaluation made by the *heuristic function* (added with the current score) was returned within the current node. Each value found would then be fed up through the game tree, selecting the maximum value for when the node represents the current player's turn and the minimum value when it is the opposing player's. When the values eventually reach the first level of the tree, the move with the *maximum payoff* would be selected.

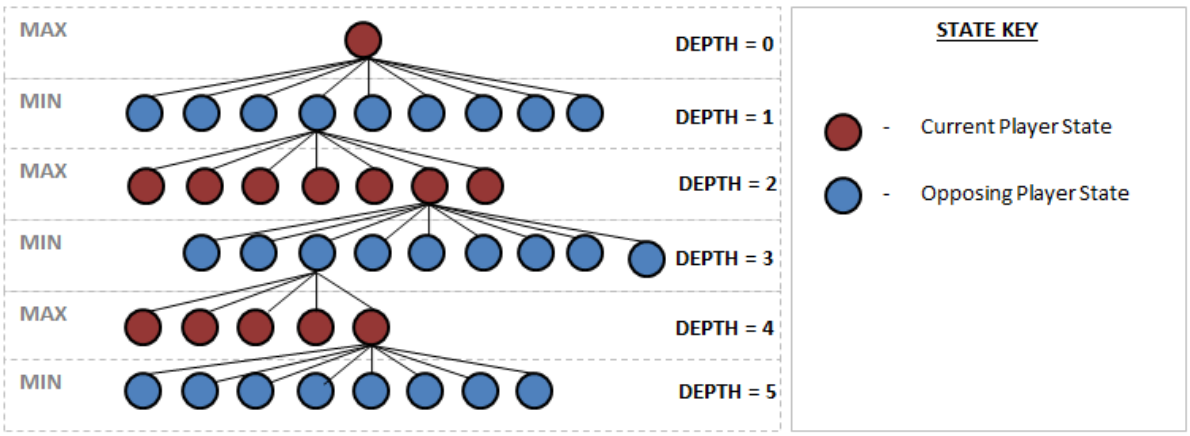


Figure 4.4 - A diagram displaying the different depth levels of a game tree, with each node representing a state for one of the two players

Upon reviewing how the *expectiminimax* algorithm performed in a typical game, I noticed that if the depth limit is greater than1, moves that involved selling cards to gain points were rarely chosen. However, the hands obtained by the artificial player would often be very valuable (e.g. hands containing five or more cards of a certain type). I realised that this was due to the *heuristic function* as it would always try to minimise the opponent's payoff by taking (and keeping) valuable sets of cards away from the market place.

The total amount of unknown cards also had an effect on how well the algorithm could predict next potential states in the game tree. This was because, the algorithm would only account for the cards known in the opposing player's hand meaning potential moves that would be possible to perform could be missed. All of these matters would need to be addressed when later developing the strategies.

4.3.5 Summary

Through using the *expectiminimax algorithm*, I have created three separate *advanced strategies* that can each be used in a single play of '*Jaipur'*.

Each strategy adopts a different depth limit yet the parameters in the heuristic evaluation function are always the same fixed values. These values were later determined in the testing stages (see *Section 5.1*). Also, for accurate state prediction, the value of the depth limit only increases once all of the opponent's cards are known. This typically happens anywhere between the 3rd to 6th player turn.

The parameters for the *heuristic evaluation function* are always the following: $k_1 = 0.58$, $k_2 = 0.38$ and $k_3 = 1.02$.

Below I have summarised how each strategy selects its next move within any state of the game.

Advanced Strategy 1

Every *move* is chosen using the *expectiminimax* algorithm.

The *depth limit* is always set to 1.

Advanced Strategy 2

Every *move* is chosen using the *expectiminimax* algorithm.

If one or more of the opponent's cards are not known, the current player possesses 3 or more goods cards of the same type, or the player possesses a total of 7 goods cards, the *depth limit* is set to 1.

Else, the *depth limit* is set to 3.

Advanced Strategy 3

Every *move* is chosen using the *expectiminimax* algorithm.

If one or more of the opponent's cards are not known, the current player possesses 3 or more goods cards of the same type, or the player possesses a total of 7 goods cards, the *depth limit* is set to 1.

Else, the *depth limit* is set to 5.

Chapter 5 – Testing the Solution

To evaluate if I had met the main aim of the project, testing how the AI performed against human players was a complete necessity. In order to do this, I first found the correct *parameter settings* used within the *heuristic function* for each *advanced strategy*. After this, each *strategy* was then played against each other to assess which one was the most successful. Using these results, I then finally conducted tests involving three separate participants to find if the AI was capable of defeating human players. Please note, as previously decided in *Section 2.3.4*, each round was treated as a single play in every test carried out.

5.1 Tuning Heuristic Parameters

Before the *advanced strategies* could be capable of making intelligent informed choices, I first needed to find the correct parameter values for the *heuristic evaluation function*. To achieve this, I decided to test *Advanced Strategy 1* against *Basic Strategy 3* using a variation of pre-selected heuristic settings over a fixed number of plays. Alternatively, I considered using two *advanced strategies* instead; however the idea was disregarded due to the possibility of ambiguous results. *Advanced Strategy 1* was chosen as it purely relied on the *heuristic function* to assess every decision it made. This could not be said for the other *advanced strategies* I had also implemented. Additionally, I chose *Basic Strategy 3* as it was the only *basic strategy* that made decisions without any elements of *randomness*. Because of this, every decision would be fully *rational* and therefore most informed when compared to the other *basic strategies* available.

When using this method to test a specific set of heuristic parameters over a fixed number of plays, the average score difference between both strategies would be calculated. Once every heuristic setting had been used, the one with the highest average score difference would be selected as the most practical to employ. Adopting this approach would still present the issue of how to find each set of parameters to test on. To resolve this, I used a sequential process that used a step size within a lower and upper bound for each parameter value. This would find a wide array of parameter combinations, thus covering a lot of potential settings. For example, if only k_1 and k_2 were being considered, and both used an upper bound of 1 and a lower bound of 0, the first four sets of parameters found using a step size of 0.5 would be the following: (k_1=0, k_2=0), (k_1=0.5, k_2=0), (k_1=1, k_2=0), (k_1=0, k_2=0.5).

Test 1 Results

To establish every heuristic setting for the first test, the following bounds and step size were used for each parameter:

- k_1: Lower Bound = 0, Upper Bound = 1
- k_2: Lower Bound = 0, Upper Bound = 1
- k_3: Lower Bound = 0, Upper Bound = 1
- Step Size = 0.2

After this, each setting determined was each used for a total of 300 plays, and every average score difference was calculated. The highest average score difference found was approximately 31.12 with the subsequent parameter settings: k_1=0.6, k_2=0.4, k_3=1 (see *Figure 5.1*). All results for *test 1* can be found in *Appendix D.1*.

Average Score Differences (k_3=1)						
k_1	**k_2 = 0**	**k_2 = 0.2**	**k_2 = 0.4**	**k_2 = 0.6**	**k_2 = 0.8**	**k_2 = 1**
0	-46.4667	-9.78	-8.39333	-5.65	-6.1	-11.4033
0.2	-25.1867	10.47	14.68333	13.44333	12.40333	9.31
0.4	-12.3567	18.01333	26.59	25.6	25.05	24.87667
0.6	2.173333	18.98667	31.12333	30.85667	28.31	23.83
0.8	7.603333	16.66333	26.58	24.1	26.27333	22.21667
1	-4.72667	3.996667	12.46667	14.53667	15.69	14.78

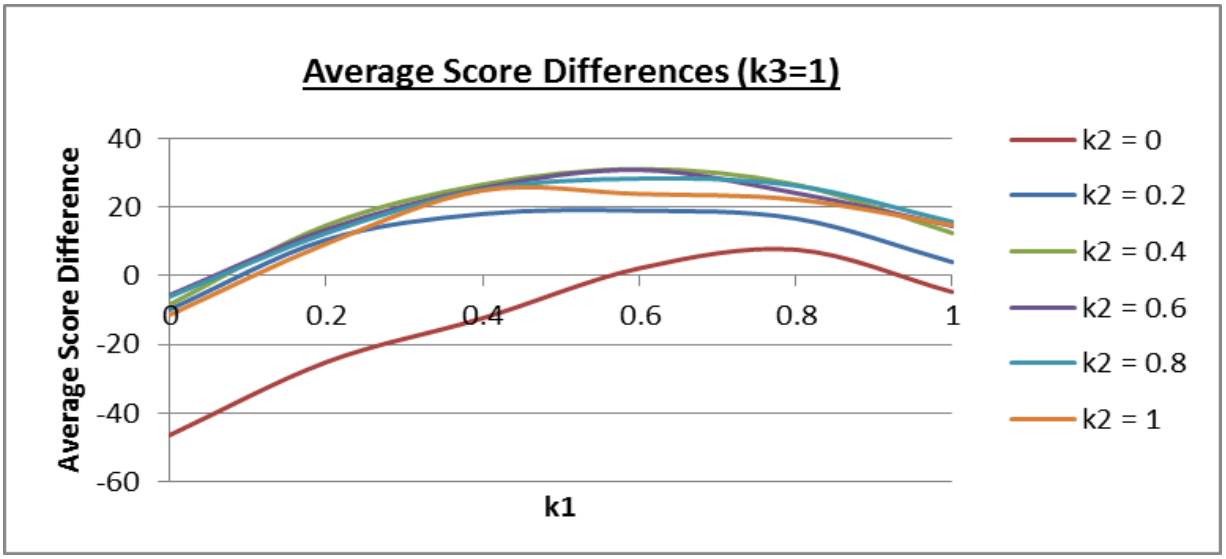


Figure 5.1 – The average score differences of heuristic parameters where step_size = 0.2, k_1_range=0-1, k_2_range=0-1 and k_3=1

Test 2 Results

To improve on this, I decided I would perform the same test again by using each newly found parameter value to calculate the new upper and lower bounds (using a range of 0.1). The step size chosen would allow each parameter to have a possible total of 6 potential new values (such as before). Thus, the following bounds and step size were used for each parameter:

- k_1: Lower Bound = 0.5, Upper Bound = 0.7
- k_2: Lower Bound = 0.3, Upper Bound = 0.5
- k_3: Lower Bound = 0.9, Upper Bound = 1.1
- Step Size = 0.04

After this, each setting was used for a total of 300 plays, and every average score difference was calculated. The highest average score difference found was approximately 34.2 with the subsequent parameter settings: k_1=0.58, k_2=0.38, k_3=1.02 (see *Figure 5.1*). All results for *test 2* can be found in *Appendix D.2*.

Average Score Differences (k_3=1.02)						
k_1	**k_2 = 0.3**	**k_2 = 0.34**	**k_2 = 0.38**	**k_2 = 0.42**	**k_2 = 0.46**	**k_2 = 0.5**
0.5	27.59333	29.45333	31.12667	30.15667	31.08333	29.18
0.54	30.91333	31.58333	30.93333	32.49	32.36	32.13333
0.58	31.59667	29.81	34.2	32.85667	33.49	31.69667
0.62	29.68667	30.04667	32.88	32.49	30.59333	33.86667
0.66	29.04	31.64667	30.21667	31.34333	32.35	31.79
0.7	28.75667	29.71667	30.27	32.01333	32.83333	30.64333

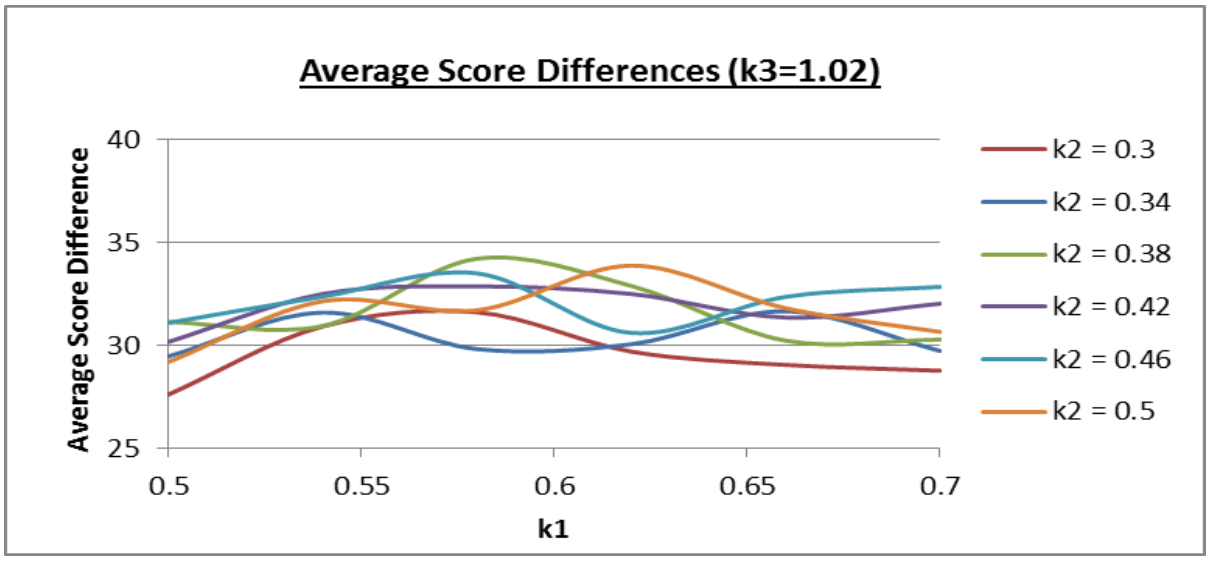


Figure 5.2 – The average score differences of heuristic parameters where step_size = 0.04, k_1_range=0.5-0.7, k_2_range=0.3-0.5 and k_3=1.02

Summary

Following the second test, I decided I would stop trying to improve upon each parameter value. Whilst observing the results, I noticed that the majority of average score differences found was roughly within a 5 point range. Because of this, if I was to search for a setting that returned an even higher score difference, the improvement in value would most likely be minimal and therefore would have very little to no impact on increasing the score obtained by the artificial player. Thus, each *advanced strategy* was to use the following *heuristic parameter settings*:

$$k_1=0.58,\ k_2=0.38,\ k_3=1.02$$

5.2 Strategy Comparison

After finding the correct settings for the heuristic function, all of the strategies (defined in *Sections 4.2.3* and *4.3.6*) were now ready to simulate an artificial player. To evaluate the performance of each one, I decided to calculate the score differences for a total of 300 plays when each strategy was used against the others available. Once the results were collected, I would then be able to judge the best strategy to use against human players. Each score difference would be calculated using the following formula:

```
SD = current_strategy_final_score - competing_strategy_final_score
```

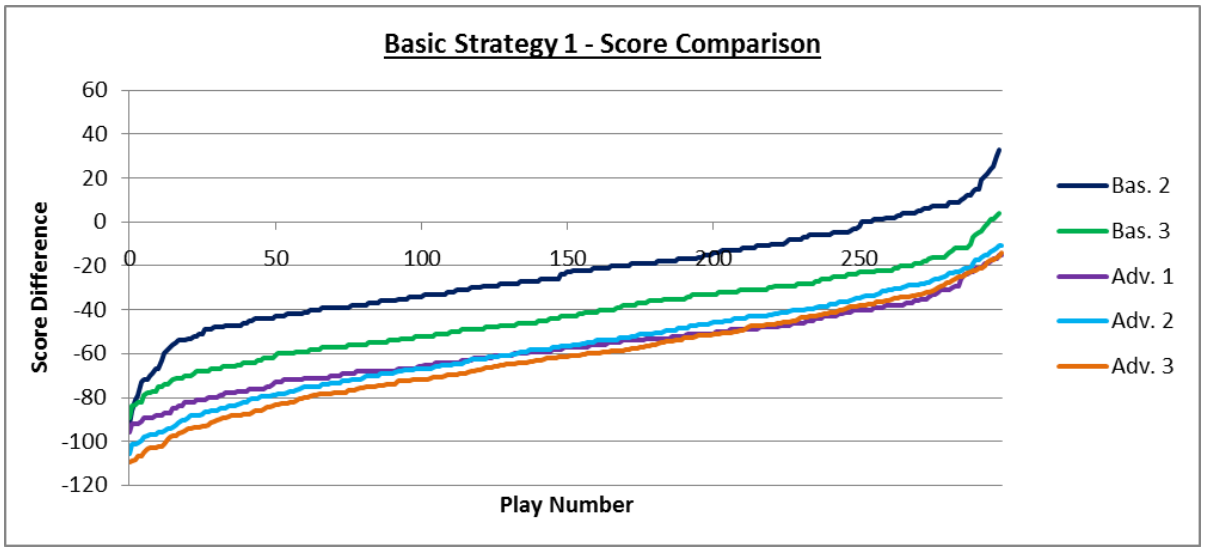


Figure 5.3 – The score differences of Basic Strategy 1 when tested against all other strategies.

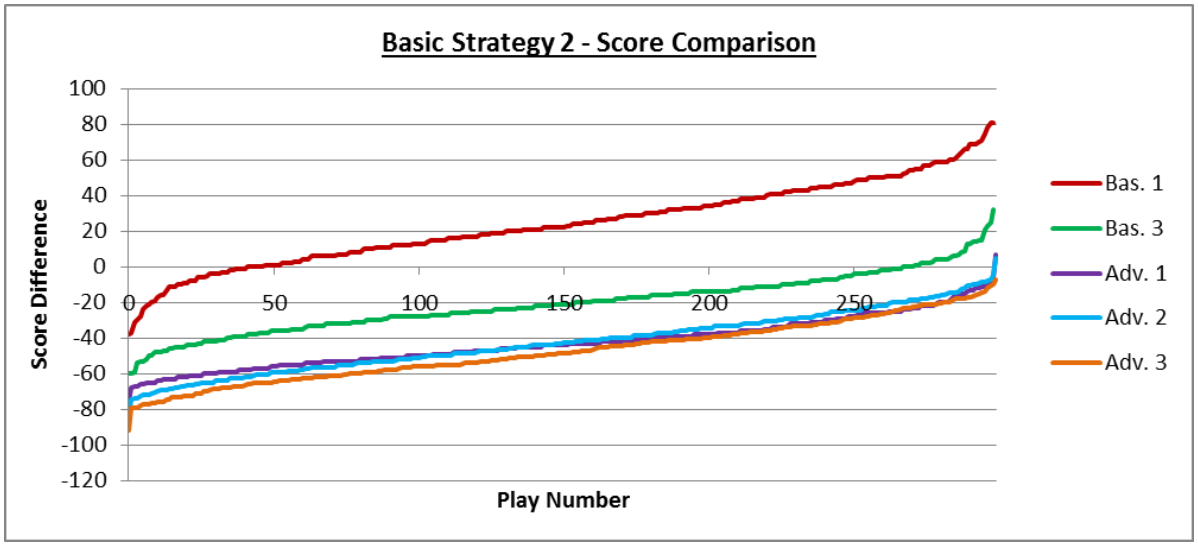


Figure 5.4 – The score differences of Basic Strategy 2 when tested against all other strategies.

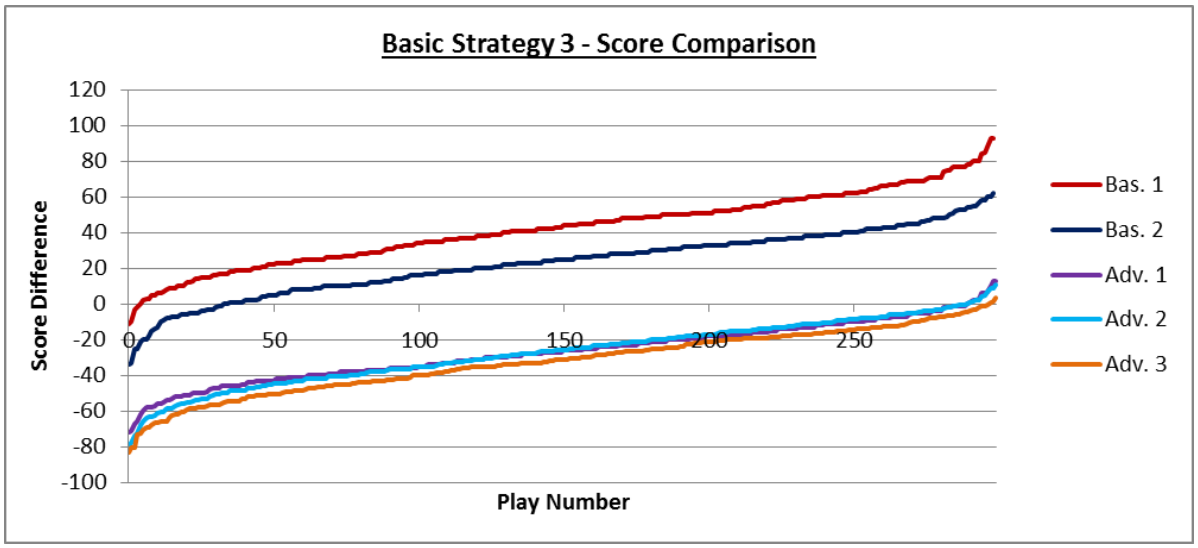


Figure 5.5 – The score differences of Basic Strategy 3 when tested against all other strategies.

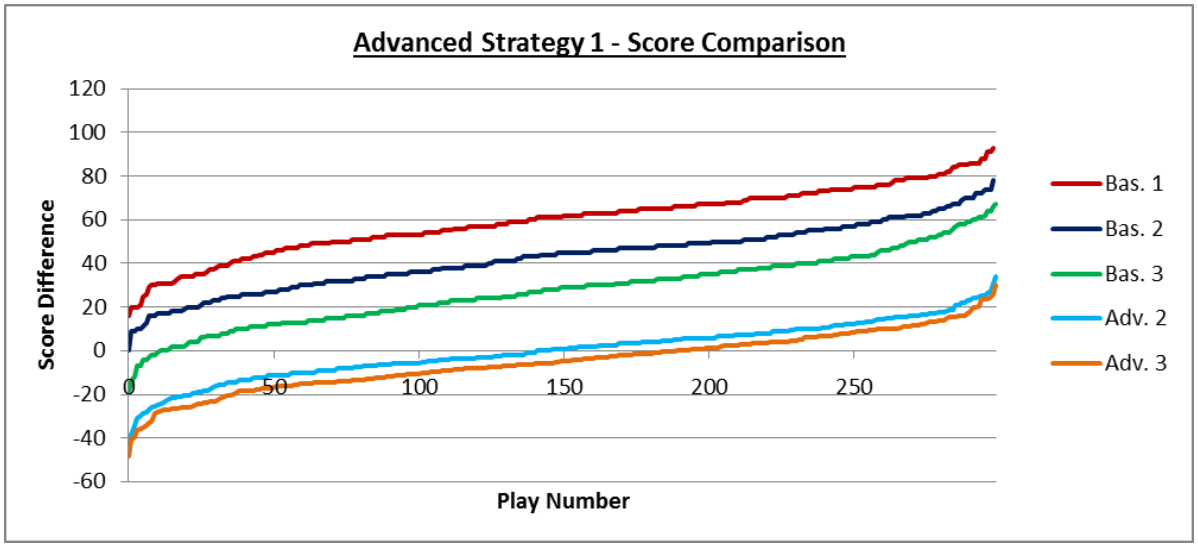


Figure 5.6 – The score differences of Advanced Strategy 1 when tested against all other strategies.

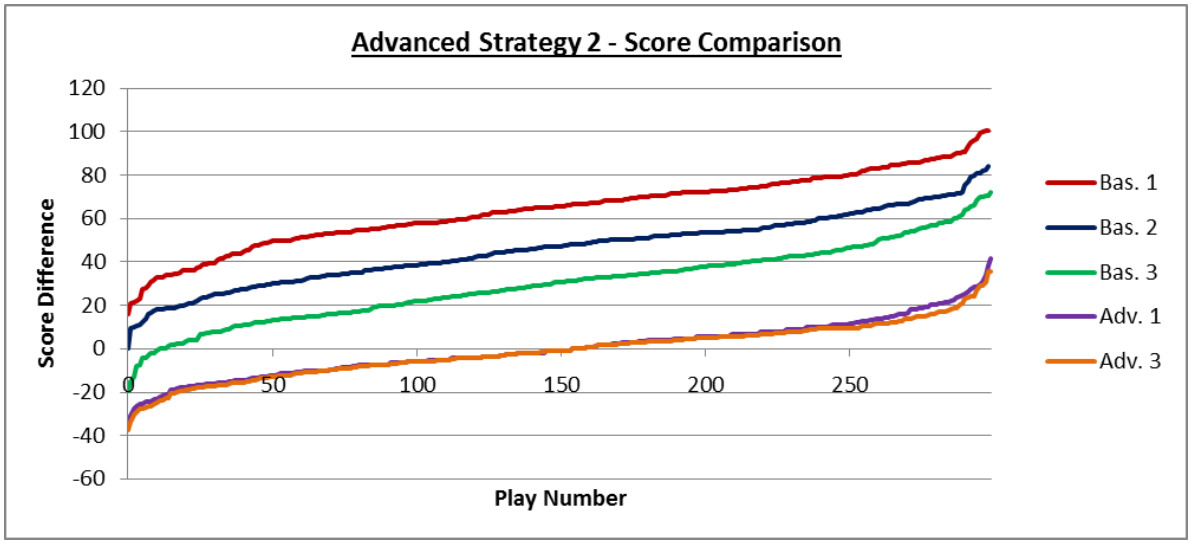


Figure 5.7 – The score differences of Advanced Strategy 2 when tested against all other strategies.

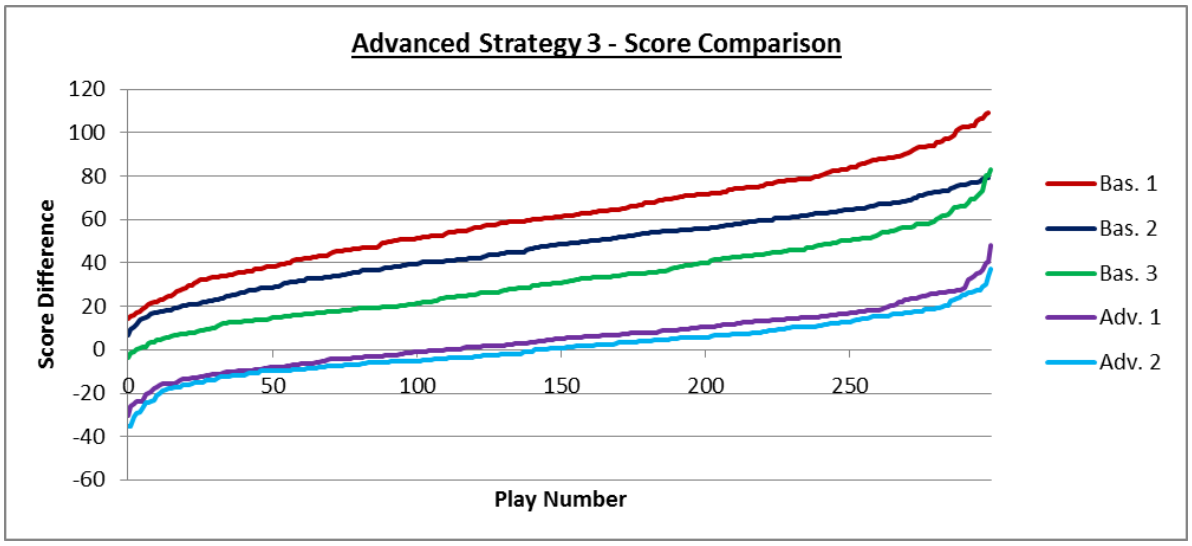


Figure 5.8 – The score differences of Advanced Strategy 3 when tested against all other strategies.

Upon reviewing the results, I first noticed the similarity of how most of the score differences for each strategy increase at a very steady rate. This is most likely due to the *zero-sum* nature of selling commodities, as once a specific token is gained by a player, the opponent will no longer be able to obtain it. However, peaks and troths are still visible at the lowest and highest score differences found within each comparison. This could be explained by the rounds that are untypical and present more advantageous situations for one player than the other due to complete luck.

The *basic strategies* have performed as expected. As the artificial player adopts less irrational choices by not selecting moves randomly, the likelihood of success seems to grow extensively. As seen in *Figures 5.3* and *5.4*, when any random approach is used against one

that is non-random (such as *Basic Strategy 3* and the *advanced strategies*), defeat is almost inevitable. As previously demonstrated in *Section 2.3.4*, the maximum number of 'selling' moves a player can perform at any given state is 5 out of a potential 37. This explains why *Basic Strategy 1* always fails as no precautions are taken to ensure that points are obtained within a single play. This is improved in *Basic Strategy 2* by prioritising 'selling' moves, yet it still usually fails as no countermeasures are taken to gain potential better future profit.

However, it is worth highlighting that *Basic Strategy 3* has shown it has the capability of beating all *advanced strategies* (see *Figure 5.5*), even if the probability of it achieving this is very low. This only strengthens the theory that random agents will most often fail against rational players, as *Basic Strategy 3* only adopts *greedy* techniques.

As the *advanced strategies* show to have an almost 100% success rate against all *basic strategies*, it can be strongly argued that using *expectiminimax* is a proficiently better method for weighing all available move choices. By analysing *Figures 5.6, 5.7* and *5.8*, it becomes evident that score differences advance by a few points as the depth limit of each search increases.

The reason why each score difference varies only slightly between each strategy can be explained through how the depth limits are selected by both *Advanced Strategy 2* and *3*. This is because the limits are only set to be greater than 1 if certain conditions are met (e.g. full knowledge of opponent's hand, each card type has a quantity less than 3 and so on). This being said, *Advanced Strategy 3* seems to be the most successful as it seems to gain a (slightly) higher score roughly 66% of the time when tested against both *advanced strategies 1* and *2*.

Initially, I desired to only pick one strategy for the next stage of testing yet all the *advanced strategies* seem to work very efficiently against the *basic strategies*, and the overall score differences for each one are also quite minor when compared. Because of this, I decided to use all three *advanced strategies* to test against the human players as I was interested to see if similar results would arise.

5.3 Human Player Tests

Using three participants, I was now ready to evaluate if the AI was capable of defeating human players. To do this, every *advanced strategy* was to be used for a total of 3 plays each and the score differences between both players would be calculated using the following formula:

```
SD = AI_player_final_score - human_player_final_score
```

Because of this, positive score differences were desired as it would mean the AI was successful in beating a human player within a specific play of the game. Each individual test would be done in an isolated environment to ensure there were no influences from outside parties when making each move choice.

Before tests could commence however, I first needed to ensure each participant was comfortable with the rules and overall gameplay in order for them to all perform at a reasonable level. This was achieved by playing the physical version of the game multiple times as it would allow each one of them to learn and develop moderate tactics in order to win (e.g. when to take/discard/sell specific goods cards). After this, I allowed each of them to become familiar with the software implementation, highlighting all of the controls and how each game component was represented.

Below, I have provided tables that display all score averages found for each *advanced strategy* tested. Within each table, the average score difference has been found for each participant against a specific strategy, and the overall average score difference has also been calculated. Using the test results alone, it can be concluded that over the 27 individual *plays* the AI had an overall 88.88% success rate.

Advanced Strategy 1 – Score Differences				
Human Player	Play 1	Play 2	Play 3	Average Score Difference
1	11	22	14	15.66
2	5	18	-10	4.33
3	28	23	22	24.33
				14.77

Advanced Strategy 2 – Score Differences				
Human Player	Play 1	Play 2	Play 3	Average Score Difference
1	-5	14	27	12
2	-3	6	19	7.33
3	14	10	31	18.33
				12.55

Advanced Strategy 3 – Score Differences				
Human Player	Play 1	Play 2	Play 3	Average Score Difference
1	17	17	26	20
2	19	30	29	26
3	25	23	21	23
				23

Chapter 6 – Conclusion

To summarise, I will conduct an evaluation of my entire project. In doing so, a verdict will be made to decide whether the final implementation meets the project aim. This will be followed by a personal reflection of my own performance and experiences, and finally a brief discussion on how the work I have produced could be extended in the future.

6.1 Aim and Objectives

To evaluate if the main aim of the project was achieved, I will review the objectives that were originally defined in *Section 1.3*.

1) **Produce a playable implementation of the card game *'Jaipur'*.**
   A platform for the game was created in *Java* which allowed for two players (as discussed in *Sections 3.1, 3.2, 3.3* and *3.4*). Each core game element was represented by a unique class and users were able to interact using a clear *text-based interface.*

2) **Perform a background study of the techniques currently used within game playing software.**
   A literature review in *Section 2.2* was conducted to discover methods previously used to simulate artificial players within games. In addition to this, extra research in *Game Theory* was conducted within *Section 2.1* followed by an in-depth analysis of '*Jaipur*' within *Section 2.3*. Doing this gave me further insight of how to approach the main task.

3) **Develop an AI-style algorithm to play the game.**
   In *Sections 4.2* and *4.3*, six separate *strategies* were implemented for the artificial player to use in a single play of the game. This included adopting both *basic* and *advanced* tactics such as *random/greedy* decision processes and an *expectiminimax* algorithm which incorporated *heuristic state evaluation* functions.

4) **Test and evaluate the algorithm against human players.**

   After all *strategies* were compared and tested in *Section 5.2*, the three that utilized the *expectiminimax* algorithm were chosen to use against human players. Results showed that each of these strategies had a near perfect success rate and allowed the AI to defeat human players (see *Section 5.3*).

5) **Compare the algorithm and test results against existing studies involving other adversarial games with AI implementations.**

   Due to time constraints, I was not able to compare my solution and test results to other similar studies. This being said, both *objectives 2* and *3* produced outputs that strengthened the final solution which exceeded the minimum requirements for essential stages of the project. Because of this reason, I feel it was acceptable to abandon this objective.

Overall, I feel the aim of this project has been met as the AI has shown it is capable of defeating human players when playing the game multiple times. However, as the game '*Jaipur*' is rather unknown, it is hard to determine if the *advanced strategies* produced would perform well against *expert level* players. For the participants used in testing, it was their first time playing the game, thus the results only prove that the strategies are effective against *rational* players that perform *moderately* at best. Nonetheless, the main aim was still achieved.

6.2 Personal Reflection

Ultimately, I have found this experience to be both challenging and difficult in certain places, however through drive and determination I have been able to produce a good final solution given the time and resources I had available to me. Over the last 12 weeks, this project has been very enjoyable and I have learnt a lot in regards to implementing an agent that can make intelligent decisions.

Saying this, my general skills in time management seemed to create a couple of recurring issues. Even though the majority of the main tasks were completed within their original timeframes, I did not account for the time needed between each one to update the written report. Because of this, on a number of occasions the report writing was put on hold and essentially had to be pushed back. This eventually led to a steadily increasing backlog and as a consequence the final objective had to be scrapped. I also did not consider other

matters outside of the project meaning the number of days initially assigned to each task were not necessarily an accurate representation of the days actually needed to complete each one.

To avoid this in future projects, I would advise that specific allocated time slots be made after each critical stage has been completed. This would ensure completing the report is a gradual process in contrast to it being left for the end of the main project scope. Additionally, for each set task I would overestimate the expected time needed to complete each one when establishing deadlines as this would account for days where no project work is done.

I was lucky that each stage and task was completed relatively on schedule even though a slightly modular process was taken in the main methodology. In future to minimise risk, instead of leaving the testing until all coding is finalised I would alternatively test each individual feature of the solution as it is implemented. Overall, I feel in order to achieve the best possible output, selecting a project which centres around something you are passionate about is key to success. Doing so will allow you to both enjoy the learning process and all stages of implementation. This is something I did, and as a result the quality of the outcome was generally good.

6.3 Future Work

Additional Testing

As stated earlier, all of the participants involved in the testing were in no way expert players in either '*Jaipur*' or other similar games. To further evaluate the performance of each existing *advanced strategy*, it'd be interesting to observe whether the same results and trends reappear whilst testing players of this level.

Additional Strategies

Improving on the techniques used within the existing *advanced strategies* may produce agents that perform better. This could include cutting down on game tree search space (through *pruning* methods) to allow deeper searches. Other *heuristic functions* could also be made to evaluate each state value by factoring different aspects of the game not previously considered such as the remaining card number.

Alternative Methods

Other approaches could be used to tackle the problem via different means. As discussed in *Section 2.2*, a strategy could be constructed through *machine learning* techniques such as

TD and *Q-Learning*. Also, instead of teaching an agent how to play a specific game it could be manufactured to play multiple unknown games using a *general game playing (GGP)* method.

List of References

[1] Newell, A., Shaw, J. and Simon, H. (1958) Chess-Playing Programs and the Problem of Complexity, *IBM Journal of Research and Development*, 2(4), pages 320-335.

[2] Samuel, A. (1959) Some Studies in Machine Learning Using the Game of Checkers, *IBM Journal of Research and Development*, 3(3), pages 210-229.

[3] Zobrist, A. (1969) A model of visual organization for the game of GO, *AFIPS Spring Joint Computing Conference 1969*, pages 103-112.

[4] Keeler, E. and Spencer, J. (1975), Optimal Doubling in Backgammon, *Operations Research*, 23(6), pages 1063-1071.

[5] Heule, M.J.H. and Rothkrantz, L.J.M. (2007) Solving Games: Dependence of applicable solving procedures, *Science of Computer Programming 67*, 1, pages 106–107, 109 and 114.

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

[7] Snidal, D. (1985). The Game Theory of International Politics. *World Politics*, 38(1), pages 25-57.

[8] Straffin, P. and Taylor, A. (1997). Mathematics and Politics. *The College Mathematics Journal*, 28(4).

[9] Sigmund, K. and Nowak, M. (1999). Evolutionary game theory. *Current Biology*, 9(14).

[10] Myerson, R. (1992). Game Theory - Analysis of Conflict. *Long Range Planning*, 25(2), page 130.

[11] Gintis, H. (2005). Behavioral Game Theory and Contemporary Economic Theory. *Analyse & Kritik*, 27(1).

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

[13] Miller, N. (2006). Introduction to Game Theory, pages 10-12.

[14] Ferguson, T. (2014). *Game Theory: Part II. Two-Person Zero-Sum Games*. 2nd edition, page 4.

[15] Leyton-Brown, K. and Shoham, Y. (2008). *Essentials of Game Theory: A Concise, Multidisciplinary Introduction*. 1st edition, pages 4-6.

[16] Cornuejols, G. and Trick, M. (1995). *Quantitative Methods for the Management Sciences*, page 132.

[17] Turocy, T. and Stengel, B. (2002). Game Theory. *Encyclopedia of Information Systems*, 2, page 22.

[18] Levin, J. (2002). *Games of Incomplete Information*. Available at: http://web.stanford.edu/~jdlevin/Econ%20203/Bayesian.pdf (Accessed 2 Apr. 2017).

[19] Kockesen, L. and Ok, E. (2007). *An Introduction to Game Theory*. 1st edition, page73.

[20] Shor, M. (2005). *Non-Cooperative Games*. Available at: http://www.gametheory.net/dictionary/Non-CooperativeGame.html (Accessed 1 Apr. 2017).

[21] Chalkiadakis, G., Elkind, E. and Wooldridge, M. (2012). Cooperative Game Theory: Basic Concepts and Computational Challenges. *IEEE Intelligent Systems*, 27(3), page 86.

[22] Hotz, H. (2006). A Short Introduction To Game Theory, pages 3-4.

[23] Mullins, Justin (2007) *Checkers 'solved' after years of number crunching*. Available at: https://www.newscientist.com/article/dn12296-checkers-solved-after-years-of-number-crunching (Accessed: 4 February 2017).

[24] Barriga, N.A. (2014) Leveraging Parallel Architectures in AI Search Algorithms for Games, *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, page 2.

[25] Javarone, M.A. (2015) Poker as a skill game: Rational versus irrational behaviours, *Journal of Statistical Mechanics: Theory and Experiment*, pages 2-4.

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

[27] Shannon. C. (1949) Programming a Computer for Playing Chess. *Philosophical Magazine*, 41, page 5.

[28] Hauk, T., Buro, M. and Schaeffer, J. (2004) *–Minimax Performance in Backgammon. *Proceedings of the Computers and Games Conference*, page 3.

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

[30] Yen, S. Chou, C. Kao, K. and Wu, I. (2014) Design and Implementation of Chinese Dark Chess Programs. *IEEE Transactions on Computational Intelligence and AI in Games*, pages 67-68.

[31] Melko, E. and Nagy, B. (2007) Optimal strategy in games with chance nodes. *Acta Cybernetica*, 18, page 172.

[32] Lamanosa, R., Lim, K., Manarang, I., Sagum, R. and Vitug, M. (2013) Expectimax Enhancement through Parallel Search for Non-Deterministic Games. *International Journal of Future Computer and Communication*, 2(5), page 466.

[33] Hsu, F. (1999) IBM's Deep Blue Chess Grandmaster Chips. *IEEE Micro*, 19, page 72.

[34] Marsland, S. (2014) *Machine Learning An Algorithmic Perspective*. 2nd Edition, pages 6-7, 302-310.

[35] Kunz, R. (2013) An Introduction to Temporal Difference Learning, page 4.

[36] Genesereth, M., Love, N. and Pell, B. (2005) General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26(2), page 63.

[37] Levine, J., Congdon, C., Ebner, M., Kendall, G., Lucas, S., Miikkulainen, R., Schual, T. and Thompson, T. (1998) General Video Game Playing.

[38] Board Game Arena. Available at: https://en.boardgamearena.com (Accessed: 1 April 2017).

[39] Gilmore, S. (2007) 'Advances in Programming Languages: Memory management', *UG4 Advances in Programming Languages*, pages 8, 10.

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/ed12j2ob/third_year_project

Appendix B

Ethical Issues Addressed

In the testing stage of my project, I used three participants to assess how well the AI competed against human players. All volunteers were provided with an information document about the project, explaining the use of the research collected. After this, they all separately signed consent forms which agreed for the results to be anonymously published within the final version of the report. These will be submitted in a detached envelope.

Appendix C

Pseudocode

C.1 Algorithm 1

```
//Input: an array C of card quantities
         an array D_tokens of diamond token values
         an array G_tokens of gold token values
         an array Sil_tokens of silver token values
         an array Clo_tokens of cloth token values
         an array Spi_tokens of spice token values
         an array L_tokens of leather token values

//Output: the current total value of array C


ALGOROTHM exact_value_of_cards(C, D_tokens, G_tokens, Sil_tokens,
                                    Clo_tokens, Spi_tokens, L_tokens)

total_card_value ← 0;

for i ← 0 to C[0] do
      total_card_value += D_tokens[i];

for i ← 0 to C[1] do
      total_card_value += G_tokens[i];

for i ← 0 to C[2] do
      total_card_value += Sil_tokens[i];

for i ← 0 to C[3] do
      total_card_value += Clo_tokens[i];

for i ← 0 to C[4] do
      total_card_value += Spi_tokens[i];

for i ← 0 to C[5] do
      total_card_value += L_tokens[i];


return total_card_value;
```

C.2 Algorithm 2

```
//Input: an array H of current hand quantities
         an array M of current market quantities
         an array S of sold card quantities


//Output: an array of all remaining card type probabilities


ALGOROTHM remaining_card_probabilities(H, M, S)

remaining_card_quant ← [ (H[0] + M[0] + S[0]), (H[1] + M[1] + S[1]),
                         (H[2] + M[2] + S[2]), (H[3] + M[3] + S[3]),
                         (H[4] + M[4] + S[4]), (H[5] + M[5] + S[5]),
                         (H[6] + M[6] + S[6]) ];

total_remaining_cards ← remaining_card_quant[0] +
                        remaining_card_quant[1] +
                        remaining_card_quant[2] +
                        remaining_card_quant[3] +
                        remaining_card_quant[4] +
                        remaining_card_quant[5] +
                        remaining_card_quant[6];

remaining_card_prob ← [remaining_card_quant[0] / total_remaining_cards,
                       remaining_card_quant[1] / total_remaining_cards,
                       remaining_card_quant[2] / total_remaining_cards,
                       remaining_card_quant[3] / total_remaining_cards,
                       remaining_card_quant[4] / total_remaining_cards,
                       remaining_card_quant[5] / total_remaining_cards,
                       remaining_card_quant[6] / total_remaining_cards];


return remaining_card_prob;
```

C.3 Algorithm 3

```
//Input: an array M of determined next market quantities
         an array P of remaining card probabilities
         a value n of the number of new cards taken from the deck


//Output: an array of the next market quantities with new card
          probability


ALGOROTHM next_market_quantities_with_prob(M, P, n)

M_with_prob ← M;

for i ← 0 to n do
      M_with_prob[0] += P[0];
      M_with_prob[1] += P[1];
      M_with_prob[2] += P[2];
      M_with_prob[3] += P[3];
      M_with_prob[4] += P[4];
      M_with_prob[5] += P[5];
      M_with_prob[6] += P[6];


return M_with_prob;
```

C.4 Algorithm 4

```
//Input: an array C of card quantities
         an array D_tokens of diamond token values
         an array G_tokens of gold token values
         an array Sil_tokens of silver token values
         an array Clo_tokens of cloth token values
         an array Spi_tokens of spice token values
         an array L_tokens of leather token values
         a heuristic weight value  k1
         a heuristic weight value  k2
         a heuristic weight value  k3

//Output: the heuristic value of array C


ALGOROTHM heuristic_value_of_cards(C, D_tokens, G_tokens, Sil_tokens,
                                   Clo_tokens, Spi_tokens, L_tokens,
                                   k1, k2, k3)

D_w ← k1; G_w ← k1; Sil_w ← k1; Clo_w ← k1; Spi_w ← k1; L_w ← k1;

if (C[0] < 2) do
      D_w ← k2;

if (C[1] < 2) do
      G_w ← k2;

if (C[2] < 2) do
      Sil_w ← k2;


total_card_value ← 0;

for i ← 0 to C[0] do
      total_card_value += D_tokens[i] * D_w;

if (C[0] % 1) > 0 do
      total_card_value += (D_tokens[C[0] - 1] * (C[0] % 1)) * D_w;

for i ← 0 to C[1] do
      total_card_value += G_tokens[i] * G_w;
```

```
if (C[1] % 1) > 0 do
      total_card_value += (G_tokens[C[1] - 1] * (C[1] % 1)) * G_w;

for i ← 0 to C[2] do
      total_card_value += Sil_tokens[i] * Sil_w;

if (C[2] % 1) > 0 do
      total_card_value += (Sil_tokens[C[2] - 1] * (C[2] % 1)) * Sil_w;

for i ← 0 to C[3] do
      total_card_value += Clo_tokens[i] * Clo_w;

if (C[3] % 1) > 0 do
      total_card_value += (Clo_tokens[C[3] - 1] * (C[3] % 1)) * Clo_w;

for i ← 0 to C[4] do
      total_card_value += Spi_tokens[i] * Spi_w;

if (C[4] % 1) > 0 do
      total_card_value += (Spi_tokens[C[4] - 1] * (C[4] % 1)) * Spi_w;

for i ← 0 to C[5] do
      total_card_value += L_tokens[i] * L_w;

if (C[5] % 1) > 0 do
      total_card_value += (L_tokens[C[5] - 1] * (C[5] % 1));

return total_card_value += (C[6] * k3);


return total_card_value;
```

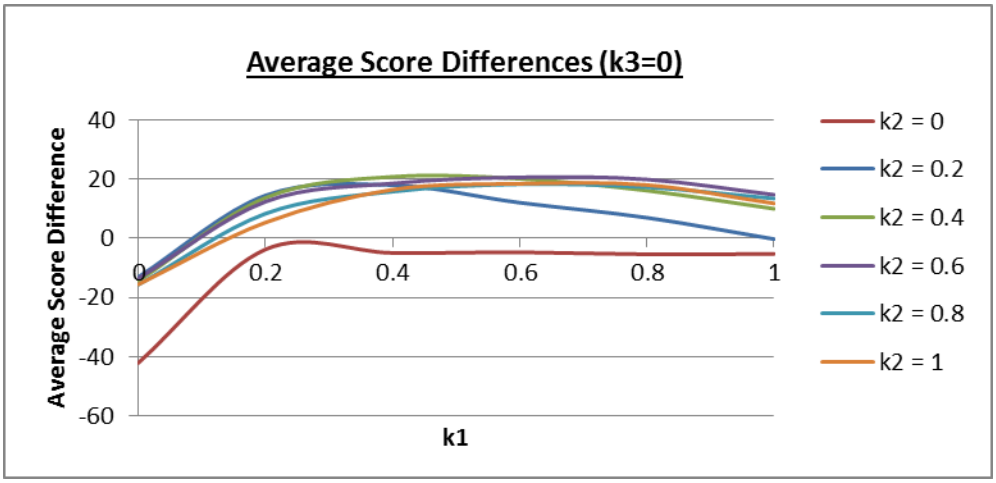
C.5 Algorithm 5

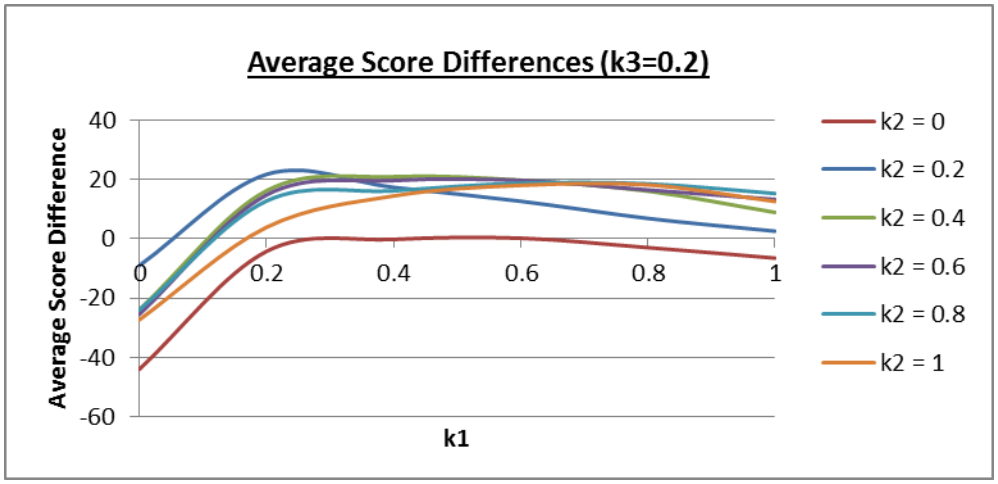
```
//Input: an array H of hand quantities
         an array M of market quantities
         a value p of points gained after sale (if one occurred)
         a value q of quantity of cards in sale (if one occurred)
         an array D_tokens of diamond token values (after sale)
         an array G_tokens of gold token values (after sale)
         an array Sil_tokens of silver token values (after sale)
         an array Clo_tokens of cloth token values (after sale)
         an array Spi_tokens of spice token values (after sale)
         an array L_tokens of leather token values(after sale)
         a heuristic weight value  k1
         a heuristic weight value  k2
         a heuristic weight value  k3

//Output: the heuristic value of current state

ALGOROTHM heuristic_value_of_state(H, M, p, q, D_tokens, G_tokens,
                                   Sil_tokens, Clo_tokens, Spi_tokens,
                                   L_tokens, k1, k2, k3)

if q == 3 do
      p += 2

if q == 4 do
      p += 5

if q == 5 do
      p += 9

hand_heuristic_value ← heuristic_value_of_cards(H, D_tokens, G_tokens,
                               Sil_tokens, Clo_tokens, Spi_tokens,
                               L_tokens, k1, k2, k3);

market_heuristic_value ← heuristic_value_of_cards(M, D_tokens,
                               G_tokens, Sil_tokens, Clo_tokens,
                               Spi_tokens, L_tokens, k1, k2, k3);

total_state_value ← p + hand_heuristic_value + market_heuristic_value;

return total_state_value;
```

Appendix D

Heuristic Parameter Tests

D.1 Test 1 Results





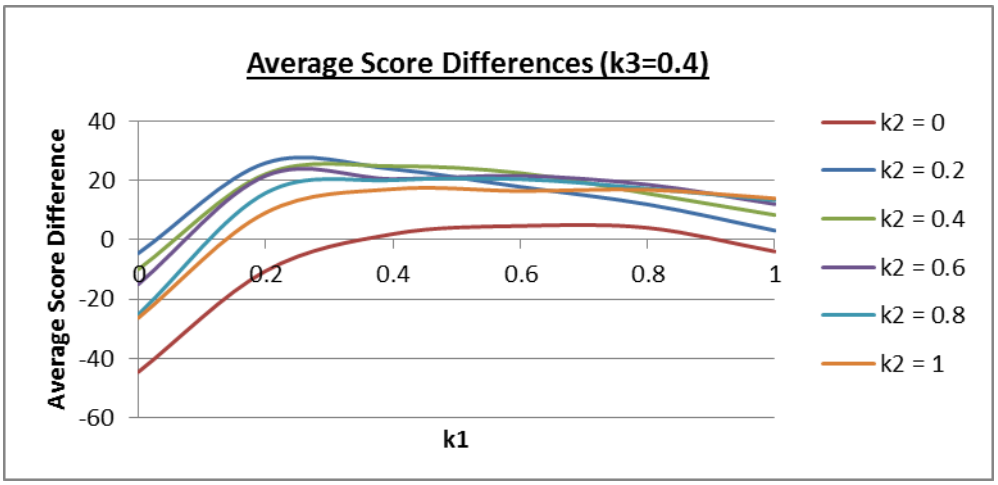
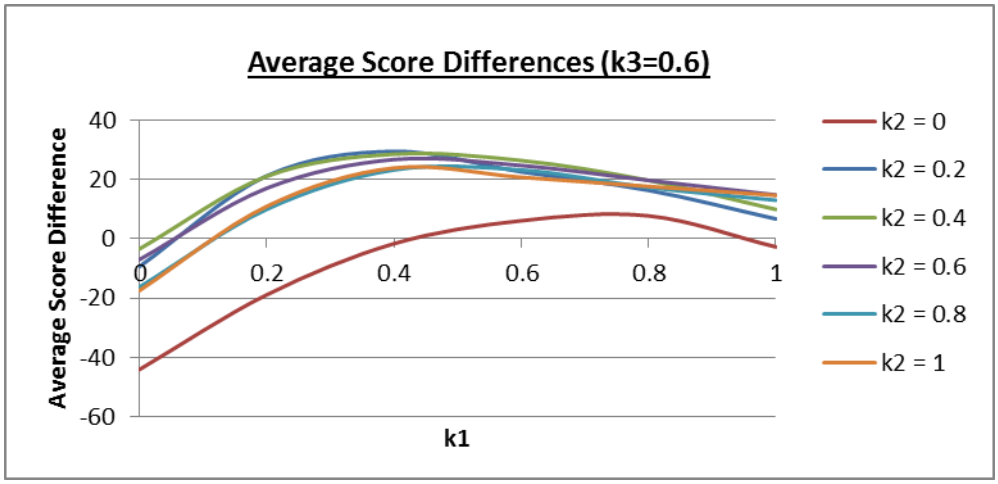
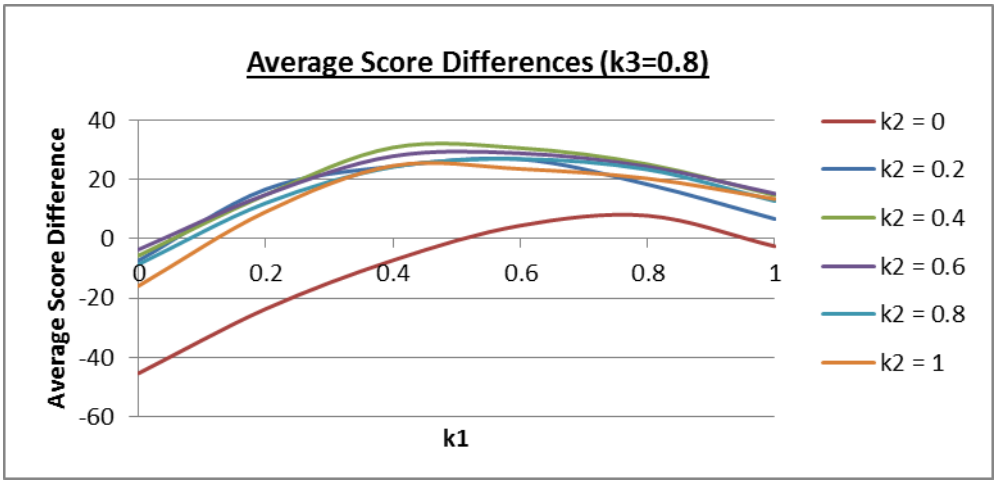


Figure D.1.1 – Graphs displaying the average score differences of heuristic parameters where step_size = 0.2, k_1_range=0-1, k_2_range=0-1 and k_3_range=0-0.4





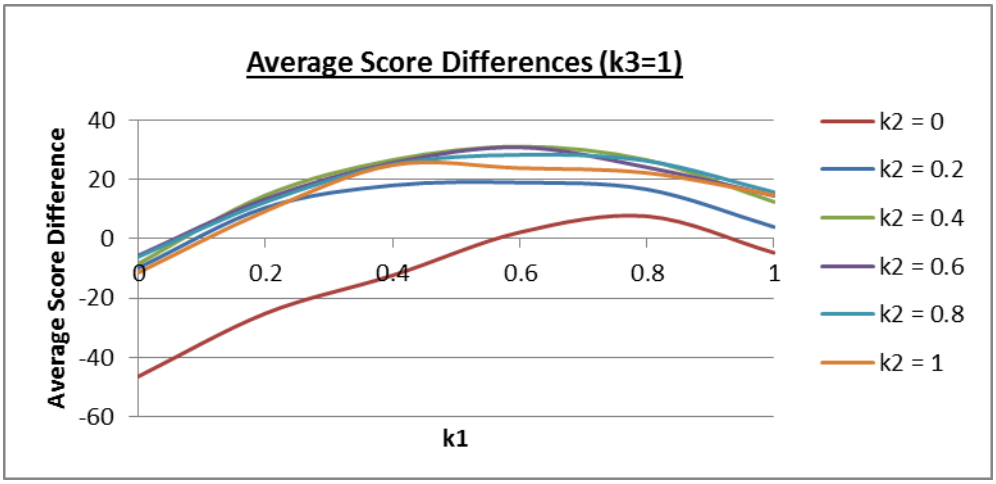


Figure D.1.2 – Graphs displaying the average score differences of heuristic parameters where step_size = 0.2, k_1_range=0-1, k_2_range=0-1 and k_3_range=0.6-1

Average Score Differences (k_3=0)						
k_1	**k_2 = 0**	**k_2 = 0.2**	**k_2 = 0.4**	**k_2 = 0.6**	**k_2 = 0.8**	**k_2 = 1**
0	-42.13	-13.0967	-14.4067	-13.5733	-15.69	-15.5867
0.2	-3.76	14.48667	13.7	12.39667	8.35	5.336667
0.4	-4.93333	17.96333	20.8	18.57	15.77333	16.45
0.6	-4.73667	12.03667	20.12	20.60333	18.27333	18.43667
0.8	-5.41	6.956667	16.12667	19.90333	17.26667	18.01667
1	-5.26	-0.26667	9.98	14.72667	13.45333	11.79667

Average Score Differences (k_3=0.2)						
k_1	**k_2 = 0**	**k_2 = 0.2**	**k_2 = 0.4**	**k_2 = 0.6**	**k_2 = 0.8**	**k_2 = 1**
0	-44.03	-9.05667	-23.9667	-25.5067	-24.0033	-27.33
0.2	-4.24	21.74	16.22	14.77	12.76333	3.88
0.4	-0.20333	17.35333	20.95667	19.65333	16.2	14.45
0.6	0.193333	12.67667	19.79	19.69667	18.76667	18.03667
0.8	-2.96	6.893333	16.08333	16.47333	18.55333	18.19333
1	-6.5	2.566667	8.906667	13.29333	15.3	12.58

Average Score Differences (k_3=0.4)						
k_1	**k_2 = 0**	**k_2 = 0.2**	**k_2 = 0.4**	**k_2 = 0.6**	**k_2 = 0.8**	**k_2 = 1**
0	-44.5733	-4.52	-9.93667	-14.9733	-25.23	-26.3267
0.2	-10.5	25.91333	22.19333	21.61	15.81	9.16
0.4	1.97	23.83667	24.83	20.5	20.11667	17.10333
0.6	4.673333	17.86333	22.53	21.57333	20.43333	16.46667
0.8	4.033333	11.94	15.61333	18.62	17.29667	16.87667
1	-3.94333	3.123333	8.36	12.05667	13.35	13.97667

Figure D.1.3 – Tables displaying the average score differences of heuristic parameters where step_size = 0.2, k_1_range=0-1, k_2_range=0-1 and k_3_range=0-0.4

Average Score Differences (k_3=0.6)						
k_1	**k_2 = 0**	**k_2 = 0.2**	**k_2 = 0.4**	**k_2 = 0.6**	**k_2 = 0.8**	**k_2 = 1**
0	-44.1	-9.37	-3.43667	-7.03333	-16.2733	-17.4733
0.2	-18.8867	21.21333	21.08667	17.06333	9.943333	10.95
0.4	-1.64333	29.48333	28.55667	26.74	23.23667	23.9
0.6	6.11	22.60333	26.42333	24.72333	23.22	20.75
0.8	7.746667	16.35	19.70667	19.73667	17.31	17.67667
1	-2.71333	6.723333	9.916667	14.81333	13.01333	14.63333

Average Score Differences (k_3=0.8)						
k_1	**k_2 = 0**	**k_2 = 0.2**	**k_2 = 0.4**	**k_2 = 0.6**	**k_2 = 0.8**	**k_2 = 1**
0	-45.3967	-7.36667	-5.69667	-3.63	-8.65667	-15.91
0.2	-23.65	16.69	14.84333	14.88667	12.04	9.086667
0.4	-7.25333	24.42	30.79667	27.87	24.24333	24.64
0.6	4.453333	26.83333	30.61	28.84667	26.89	23.57667
0.8	7.79	18.34667	25.15333	24.36	23.36333	20.33
1	-2.54	6.703333	14.69	15.23	12.76	13.42333

Average Score Differences (k_3=1)						
k_1	**k_2 = 0**	**k_2 = 0.2**	**k_2 = 0.4**	**k_2 = 0.6**	**k_2 = 0.8**	**k_2 = 1**
0	-46.4667	-9.78	-8.39333	-5.65	-6.1	-11.4033
0.2	-25.1867	10.47	14.68333	13.44333	12.40333	9.31
0.4	-12.3567	18.01333	26.59	25.6	25.05	24.87667
0.6	2.173333	18.98667	31.12333	30.85667	28.31	23.83
0.8	7.603333	16.66333	26.58	24.1	26.27333	22.21667
1	-4.72667	3.996667	12.46667	14.53667	15.69	14.78

Figure D.1.4 – Tables displaying the average score differences of heuristic parameters where step_size = 0.2, k_1_range=0-1, k_2_range=0-1 and k_3_range=0.6-1

D.2 Test 2 Results

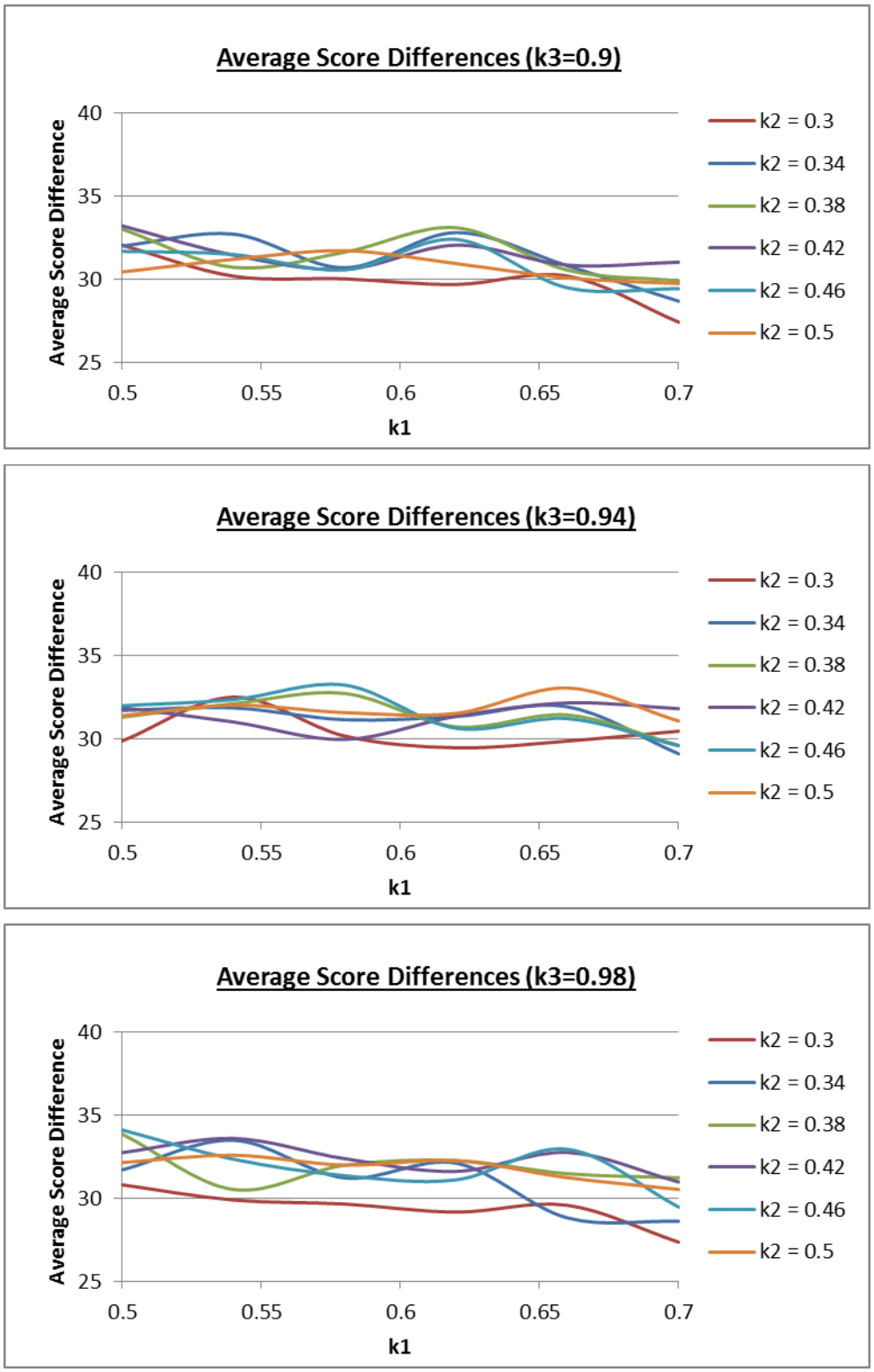
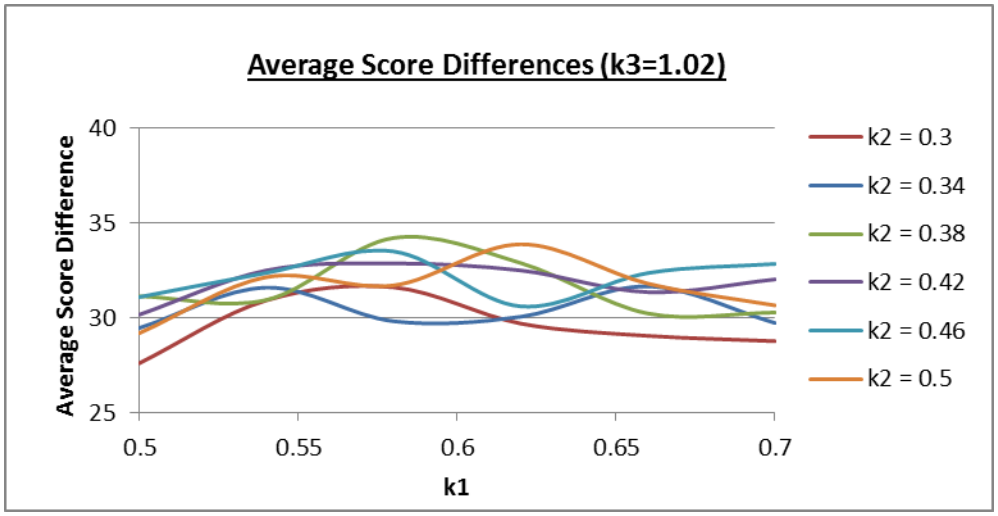
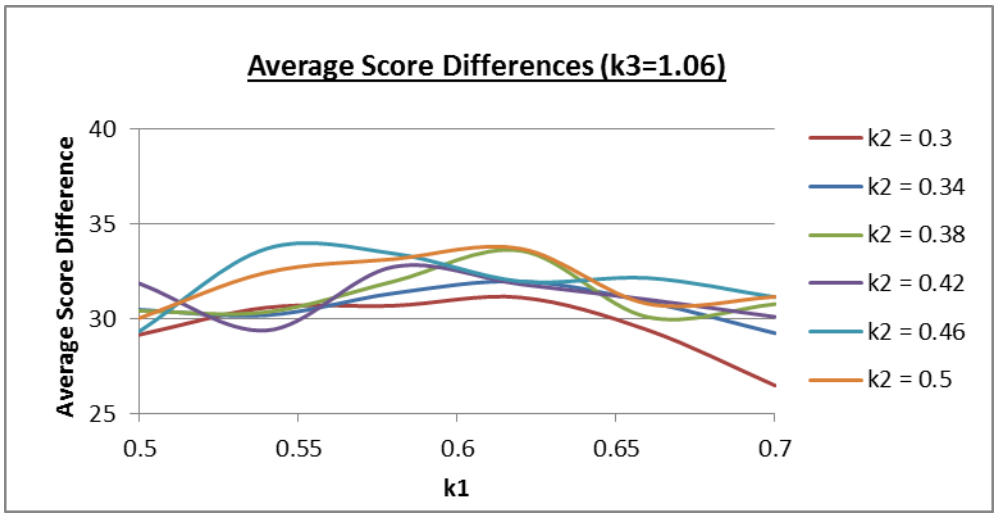


Figure D.2.1 – Graphs displaying the average score differences of heuristic parameters where step_size = 0.04, k_1_range=0.5-0.7, k_2_range=0.3-0.5 and k_3_range=0.9-0.98





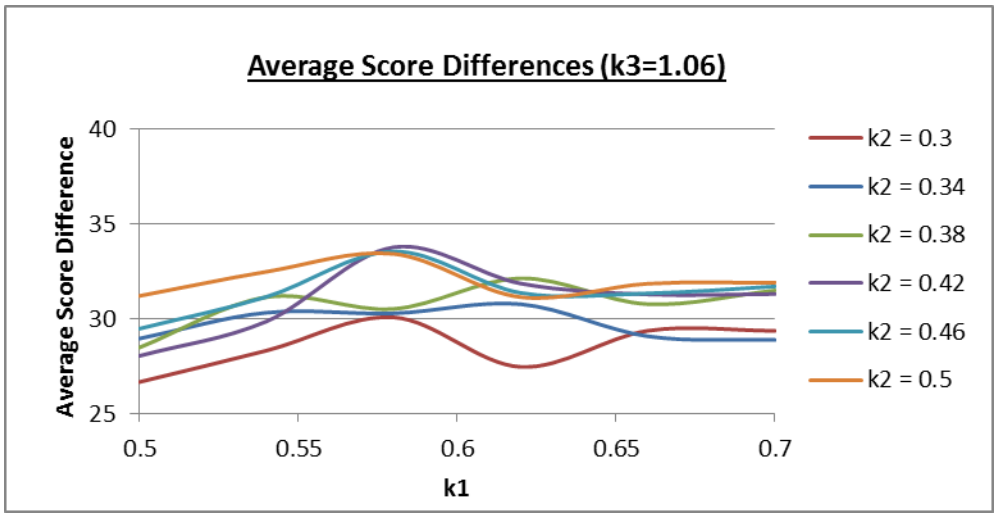


Figure D.2.2 – Graphs displaying the average score differences of heuristic parameters where step_size = 0.04, k_1_range=0.5-0.7, k_2_range=0.3-0.5 and k_3_range=1.02-1.1

Average Score Differences (k_3=0.9)						
k_1	**k_2 = 0.3**	**k_2 = 0.34**	**k_2 = 0.38**	**k_2 = 0.42**	**k_2 = 0.46**	**k_2 = 0.5**
0.5	32.05	31.99667	33.01333	33.21333	31.66667	30.43
0.54	30.18	32.71	30.70667	31.45	31.48667	31.19
0.58	30.01333	30.68667	31.62333	30.58667	30.56	31.71
0.62	29.67667	32.8	33.09333	32.04667	32.39	30.93667
0.66	30.18333	30.8	30.54333	30.84667	29.47667	30.07
0.7	27.41	28.67	29.89667	31.02	29.42333	29.73

Average Score Differences (k_3=0.94)						
k_1	**k_2 = 0.3**	**k_2 = 0.34**	**k_2 = 0.38**	**k_2 = 0.42**	**k_2 = 0.46**	**k_2 = 0.5**
0.5	29.86	31.70333	31.29667	31.87667	31.98333	31.38667
0.54	32.50333	31.85667	32.11	31.01	32.35333	32.01667
0.58	30.16333	31.14333	32.72333	29.96	33.22333	31.57333
0.62	29.45667	31.34	30.71667	31.35333	30.63333	31.52667
0.66	29.86	31.92	31.42667	32.15667	31.21333	33.05
0.7	30.45667	29.09333	29.6	31.81	29.57667	31.06333

Average Score Differences (k_3=0.98)						
k_1	**k_2 = 0.3**	**k_2 = 0.34**	**k_2 = 0.38**	**k_2 = 0.42**	**k_2 = 0.46**	**k_2 = 0.5**
0.5	30.80667	31.69333	33.86333	32.74333	34.11	32.14333
0.54	29.88667	33.48667	30.52333	33.59667	32.34667	32.58
0.58	29.64667	31.22	31.98667	32.38	31.35667	31.99333
0.62	29.16	32.10667	32.26333	31.61	31.10333	32.25667
0.66	29.57333	28.81333	31.47333	32.75	32.94667	31.23333
0.7	27.35	28.60667	31.23	30.97333	29.46333	30.52667

Figure D.2.3 – Tables displaying the average score differences of heuristic parameters where step_size = 0.04, k_1_range=0.5-0.7, k_2_range=0.3-0.5 and k_3_range=0.9-0.98

Average Score Differences (k_3=1.02)						
k_1	**k_2 = 0.3**	**k_2 = 0.34**	**k_2 = 0.38**	**k_2 = 0.42**	**k_2 = 0.46**	**k_2 = 0.5**
0.5	27.59333	29.45333	31.12667	30.15667	31.08333	29.18
0.54	30.91333	31.58333	30.93333	32.49	32.36	32.13333
0.58	31.59667	29.81	34.2	32.85667	33.49	31.69667
0.62	29.68667	30.04667	32.88	32.49	30.59333	33.86667
0.66	29.04	31.64667	30.21667	31.34333	32.35	31.79
0.7	28.75667	29.71667	30.27	32.01333	32.83333	30.64333

Average Score Differences (k_3=1.06)						
k_1	**k_2 = 0.3**	**k_2 = 0.34**	**k_2 = 0.38**	**k_2 = 0.42**	**k_2 = 0.46**	**k_2 = 0.5**
0.5	29.15	30.47	30.40333	31.85667	29.32333	30.02333
0.54	30.58	30.17667	30.34	29.37667	33.68667	32.43
0.58	30.68	31.33667	31.96	32.73	33.43333	33.13667
0.62	31.11333	31.96	33.58	31.81	31.96	33.68667
0.66	29.39333	30.88333	30.07667	31.00667	32.14	30.78
0.7	26.47333	29.22667	30.77	30.09333	31.13	31.15

Average Score Differences (k_3=1.1)						
k_1	**k_2 = 0.3**	**k_2 = 0.34**	**k_2 = 0.38**	**k_2 = 0.42**	**k_2 = 0.46**	**k_2 = 0.5**
0.5	26.65333	28.94333	28.47333	28.03	29.46667	31.19333
0.54	28.31	30.31333	31.13333	29.85	31.15	32.47
0.58	30.06667	30.27667	30.51	33.76	33.54333	33.41667
0.62	27.45333	30.76	32.12667	31.85	31.35667	31.13333
0.66	29.36	29.06667	30.76333	31.27333	31.33	31.84333
0.7	29.35667	28.87333	31.48667	31.29667	31.7	31.89667

Figure D.2.4 – Tables displaying the average score differences of heuristic parameters where step_size = 0.04, k_1_range=0.5-0.7, k_2_range=0.3-0.5 and k_3_range=1.02-1.1