GAME PLAYING: BOARD OR CARD GAMES ”OH HELL!”

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Student id: 10895352

Department of Computer Science

Contents

Abstract **6**

Declaration **7**

Copyright **8**

1 Introduction **9**

1.1 Aims and Objectives 9

1.2 Outlining the Rules of “Oh Hell!” 10

1.2.1 Preliminaries 10

1.2.2 Play 10

1.2.3 Bidding 10

1.2.4 Scoring 10

1.2.5 Special Rounds 11

1.3 Organisation of Report 11

2 Background and Literature **12**

2.1 Why ”Oh Hell!”? 12

2.2 Self-Play Heuristic Optimisation 13

2.3 Monte Carlo Tree Search 13

2.4 Information Set Monte Carlo Tree Search 14

2.5 AI in Card Games 15

3 Design **17**

3.1 Making the Game 17

3.1.1 Classes and parameters 17

3.1.2 User interface and interaction 21

3.2 Informed Agent 23

3.2.1 The primary challenges in ”Oh Hell!” 23
3.2.2 Informed agent version 1 - naive strategy 23
3.2.3 Informed agent version 2 - counting cards 28
3.2.4 Informed agent version 3 - a weighted combination of multiple heuristics 30
3.3 Monte Carlo Agent 34
3.3.1 The motivation behind Monte Carlo Tree Search (MCTS) . . 34
3.3.2 The challenges of applying MCTS to ”Oh Hell!” 37
3.3.3 Designing the Monte Carlo Agent 39
3.3.4 Optimisations 44

4 Implementation 46
4.1 Deciding the Winner of a Trick 46
4.2 Informed Player’s Self-Play 48
4.3 ISMCTS Determinisation Step 50

5 Experimentation and Evaluation 52
5.1 Random Agent Evaluation 52
5.2 Informed Agent Evaluation 57
5.3 Monte Carlo Agent Evaluation 61

6 Conclusions 69
6.1 Summary of Work and Discussion of Results 69
6.2 Further Work 70

Bibliography 71

A MCTS Implementation 74

B Unit tests 84

Word Count: 13588

List of Figures

2.1 Diagram of MCTS algorithm's 4 phases [BPW$^+$12] 14

3.1 Class diagram of "Oh Hell!" (excluding the other agents) 20
3.2 Flow diagram showing the phases of the game which the game class outlines 21
3.3 Graphical user interface design 22
3.4 Flow chart of a naive approach to the bidding problem 24
3.5 Flow chart of a naive approach to the playing to bid problem 27
3.6 Diagram of an example of the winning probability heuristic 29
3.7 Diagram of an example of the same suit heuristic 30
3.8 Diagram of an example of the follow-suit heuristic 31
3.9 Diagram of an example of the trump-probability heuristic 32
3.10 Diagram of an example of the suit-safety heuristic 32
3.11 Flow diagram of the weight optimisation algorithm using Optuna . . . 34
3.12 Flow diagram outlining the phases of the MCTS algorithm 36
3.13 Flow diagram outlining the phases of the ISMCTS algorithm 38
3.14 Diagram of an example of a state in the game tree 42
3.15 Diagram of an example of a game tree showing the wins and visits of each player and the card that yielded this 43

4.1 Image showing the transitive relationship between cards beating each other 46
4.2 Screenshot of the evaluation function for self-play along with the terminal output 49

5.1 Example of how cards of the same suit as the lead card must be played 53
5.2 Graph showing the standard deviation of 4 different random agents' scores playing against each other over different numbers of samples . 54

5.3 Graph showing the average change in standard deviation over all 4 agents over different numbers of samples 54
5.4 Graph showing the scores of random agents playing against other random agents over 300 simulations 56
5.5 Graph showing the number of bids made by random agents playing against other random agents over 300 simulations 57
5.6 Example of how the play option algorithm wins when under bid if it can 58
5.7 Graph showing the scores of informed agents playing against other informed agents over 300 simulations 58
5.8 Graph showing the bids made by informed agents playing against other informed agents over 300 simulations 59
5.9 Graph showing the scores of an informed agent playing against random agents over 300 simulations 60
5.10 Graph showing the bids made by an informed agent playing against random agents over 300 simulations 61
5.11 Graph showing the scores of a Monte Carlo agent playing against random agents over 20 simulations 62
5.12 Graph showing the bids made by a Monte Carlo agent playing against random agents over 20 simulations 63
5.13 Results from question 1 of the user survey asking how long participants have played the game 65
5.14 Graph showing the scores of participants against each type of agent over 22 samples 66
5.15 Graph showing the average scores of agents against participants over 22 samples 66
5.16 Graph showing the difference between scores of the participant and the highest scoring agent in each game 67
5.17 Histogram of the scores of participants 68

Abstract

GAME PLAYING: BOARD OR CARD GAMES
"OH HELL!"
William Sarsfield
A dissertation submitted to The University of Manchester
for the degree of Master of Science, 2024

British “Oh Hell!” is a simpler variation of the trick-taking game Bid Whist, which is quick to learn but difficult to master, containing both elements of chance and strategy. In this report, I aim to design, implement and test software for human players to play “Oh Hell!” and propose different methods for making artificial intelligence opponents that play the game to a high level. The project is made up of 3 main phases, firstly designing and implementing the game and building an agent that plays legal random moves, then taking a heuristic approach based on human-inspired strategies to make an informed agent, and finally, creating a Monte Carlo agent that attempts to search through states of the game and produce the best move given its knowledge.

The paper precisely describes the process taken to create each agent and the challenges that were overcome during the decision-making associated with design and implementation. Once the agents are finished they are evaluated against the other agents as well as real players who took part in my user evaluation. The data gathered suggests that the knowledge-based heuristic approach has a characteristic strategy which is recognisable by real players, whilst the Monte Carlo agent is proven to be the best agent against real players. The level of expertise of the agent is deemed uncertain due to a small, biased sample of “Oh Hell!” players that participated in the user evaluation.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Chapter 1

Introduction

1.1 Aims and Objectives

This project aims to design, implement, and experiment with artificial intelligence agents that show good performance against other agents and real players in the game "Oh Hell!". The best of the agents should show an expert level of play in comparison to real players. Secondly, I wish to explore the use of different artificial intelligence techniques in the creation of my agents and show this clearly with the different levels of agents. The objectives I set to achieve these aims are as follows.

1. Implement a working game of "Oh Hell!" which asks an agent class for its chosen move given a list of possible moves. Making this working game requires the creation of a default agent that chooses a legal move at random.

2. Create a primitive graphical user interface so that games between agents can be viewed visually to give me a direct understanding of how the games are being played.

3. Add an interactive element to this user interface so that a real player can play against the agents.

4. Make an agent that played the game only utilizing a set of heuristics that could be fine-tuned through self-play.

5. Create an agent that implements the Monte Carlo Tree Search algorithm [BPW+12].

6. The final objective is to produce accurate metrics of performance, including evaluations from simulated games and games against real players.

1.2 Outlining the Rules of “Oh Hell!”

1.2.1 Preliminaries

“Oh Hell!” can be played with 3 to 7 players, with a pack of 52 playing cards ranking AKQJT98765432 [Par08]. Players agree upon how many cards they wish to be dealt in the first round. After each round, the number of cards dealt to each player is reduced by 1 until each player has 1 card. After the 1 card round, the number of cards dealt is increased until the starting hand size is reached, which is typically when the game ends. The cards are dealt face down to each player. Once the cards are dealt, each player may look at their hand. The trump suit is specified before the bidding phase.

1.2.2 Play

Starting from the left of the dealer, the first player can play any card in their hand, every other player must then play a card that follows suit if they have a card in that suit. If they do not have a card of the same suit, they may play any card in their hand. Once each player has played a card, this marks the end of a trick. The winner of the trick is given to the player that played the highest-value card of the same suit, any cards not of the same suit cannot win, unless it is a trump card. The trump suit is decided at the beginning of the round (chosen by rotation through a specified sequence of suits e.g. hearts, diamonds, spades, clubs) and if cards of this suit are played on another suit, it wins the trick. If multiple players play a trump card, the highest-valued trump card wins. Tricks are played until there are no cards left.

1.2.3 Bidding

Before the playing stage, each player announces in turn, from the left of the dealer, exactly how many tricks they are going to take by the end of the round. The player who dealt will also be the last player to bid, this player must make sure their bid does not make the total bidding add up to the number of cards in hand. This ensures that at least 1 person will not make their bid each round.

1.2.4 Scoring

Once a round is finished, the number of tricks won by each player is added to their score. A bonus of 10 points is awarded to players that make their bid.

1.2.5 Special Rounds

Some notable special rounds can be played in “Oh Hell!”. These include half-blind rounds where the trump suit is revealed after the bids are made. Full-blind rounds where players bid before looking at their cards. No-trump rounds where the rounds are played without a trump suit. I did not implement these rounds into the game for simplicity, but with more time these rounds could be implemented as an extension to the game.

1.3 Organisation of Report

Chapter 2 is titled Background and describes the concepts necessary to understand the motivation behind choosing “Oh Hell!” and the technical elements used to build the different agents. The design section follows in Chapter 3 and details the process of creating the game in addition to the design of the different agents and the user interface. After this, a description of the decisions taken and the challenges faced when programming different notable areas of the code is given in Chapter 4. Finally, the evaluation and experimentation Chapter tries to gauge the performance of the different agents through rigorous experimentation including self-play and user evaluation. My critical opinions and final thoughts on the agents and the game then conclude the report.

Chapter 2

Background and Literature

2.1 Why ”Oh Hell!”?

The game ”Oh Hell!” is in the family of trick-taking games and is derived from the game Whist. ”Oh Hell!” was first described by B.C. Westall around 1930 [Par08] and can be described as a simplification of Bid Whist. Bid Whist follows a similar set of rules as described in the introduction section, with the main difference being bidding. The bidding in Bid Whist is more complex and follows a similar system to Bridge: ”Starting with the left of the dealer, the player declares how many tricks beyond six tricks that they can take with their partner, as well as the trump suit, and uptown/downtown, which signals whether high or low cards will win a trick, respectively.” [Ope08]. I believe this bidding system introduces more uncertainty in the bidding phase. An increase in uncertainty means a less deterministic game overall, which removes the opportunity for agents to display good strategy. This is evident through deterministic games like Go and Chess where agents like AlphaZero were created that produced superhuman-level play [HWCH18]. Because of the deterministic nature of the game, if an agent is better than another it should consistently win. This is not the case in games like ”Oh Hell!” where chance arises from the distribution of cards among the other players and the cards that are not in play. It is clear from this that minimising the level of uncertainty allows the agents to be evaluated more accurately over fewer samples because of fewer chance elements. This is one of the primary reasons dictating my decision to use ”Oh Hell!” instead of the more popular Bid Whist. Beyond this, I decided on ”Oh Hell!” because I enjoy playing it with friends and family, and I find the balance of strategy, chance, and simplicity very charming.

2.2 Self-Play Heuristic Optimisation

A heuristic is a simple, and efficient problem-solving strategy that speeds up or finds the solution to a decision-making problem; they typically come from intuition and experience. Given a set of heuristics, maximise an agent's performance by optimising the weights assigned to each heuristic. This forms the basis for the type of optimisation problems that are mentioned in this report. This is similar to hyperparameter-tuning problems seen in machine learning where we pass weights into a model in an attempt to minimise a loss function. In this project, I similarly optimise parameters in a simulation-based optimisation problem. This is where weights are applied to the heuristics in the agents, and iteratively through self-play, it maximises the agent's average performance over many simulated games. I used a library called Optuna which uses a combination of random search and Bayesian optimisation [ASY$^+$19a] to incrementally improve the score of the agents by selecting a subset of the parameters to perturb over many different trials. Each trial is evaluated by pitting the agent with the newly altered parameters against 3 agents with the same parameters as the current highest-scoring agent. Self-play refers to agents playing each other with different weighted parameters that affect how they play. By getting these different agents to play against each other, the winner from many different games can be selected as the base for new agents in another round of self-play.

2.3 Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) algorithm involves incrementally growing a game tree whilst also evaluating the payoff at each state. Game trees consist of nodes which represent states, and edges which represent moves to transition between states. The root of the tree is the current state of the game. MCTS first applies a selection policy (tree policy) recursively to descend through the tree until the expandable node with the highest UCT value is reached. Expandable nodes are those which have possible transitions that do not yet lead to a node. Next, the algorithm expands the selected node by choosing a random move that transitions to the newly expanded node. Once we have an expanded node we simulate a game from this game state until the game's termination according to a default policy. The default policy dictates how agents play in simulations; this is often "random playouts". The simulation's result is then back-propagated up the tree through the selected nodes that lead to that result [BPW$^+$12].

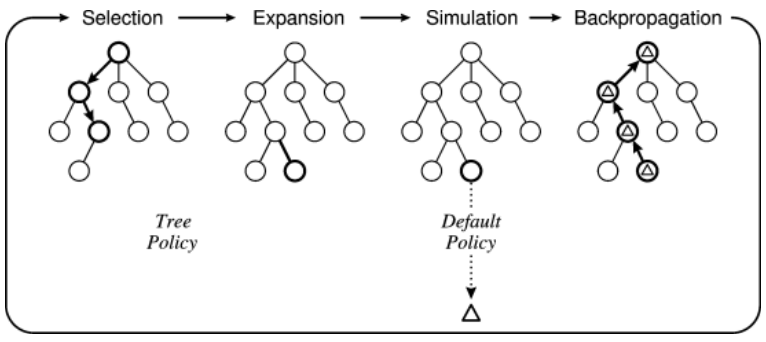


Figure 2.1: Diagram of MCTS algorithm's 4 phases [BPW$^+$12]

2.4 Information Set Monte Carlo Tree Search

Information Set Monte Carlo Tree Search (ISMCTS) is a variation of MCTS which is a solution to the problem of representing game states in games of imperfect information. Games of imperfect information are those in which the player does not know which game state they are currently in; for example, in most card games, the player does not know which cards their opponents possess. Imperfect information poses a challenge for MCTS since moves' values cannot accurately be measured since opponents' responses may vary depending on the information they have [DR11]. The solution to this problem lies in the use of information sets, these are the set of possible states that are consistent with the player's current information. In the context of most card games, an information set would include every single state where the hand of the player is consistent and all the possible distribution of cards in the hands of the other players given the remaining unseen cards. Given the context of information sets, ISMCTS utilises this by adding another step to the MCTS algorithm called determinisation [CWP15]. This occurs before the other steps. By choosing a random state in the information set, the MCTS algorithm can be applied to the deterministic game state to calculate the value of each move. When this is repeated to sample multiple states in the information set, an accurate evaluation of the possible moves can be calculated over time.

Like the regular MCTS algorithm, ISMCTS relies on many iterations to converge to the best move. However, this introduces a trade-off between a faster learning rate and a better model. An increase in the requirement of iterations means longer thinking time is necessary for an accurate evaluation. This therefore adds a hyperparameter to the algorithm, while samples of the information set can be repeated until the time limit expires, the number of iterations per determinisation must be explicit [CWP15]. Every state in the information set cannot be sampled within a reasonable time. For card games like “Oh Hell!”, there is often a large number of possible permutations of cards that could be in the opponent’s hand, the time limit imposed is for user experience. It ensures the user does not have to wait a long time between moves.

2.5 AI in Card Games

Many card games exist with similar attributes as “Oh Hell!”, games which have imperfect information that usually consists of 3 or more players. In this section, I will cover a range of techniques that could be employed to create different agents for “Oh Hell!” based on the fact they have been used to create agents for other similar card games.

The game of Hearts is a similar trick-taking game with a different scoring system. Sturtevant et al. [SW07] made an AI for Hearts that utilised Stochastic Linear Regression and Temporal Difference Learning (TDL). TDL is a type of reinforcement learning which takes an input state (game state) and returns an output state (an action), the environment then rewards the agent based on how “good” or “bad” their output was [T+95]. Since rewards are usually delayed until a score is produced, TDL must solve the “‘temporal credit assignment” problem to decide which actions should be given which reward. This approach could be applied to “Oh Hell!” where agents are rewarded at the end of each round based on their score for that round. The problem with this however is going about solving the temporal credit assignment problem. It is difficult to know which moves caused a player to fail to make their bid, making this approach less appealing.

Another method that an agent could utilise is Counterfactual Regret Minimisation. This is a self-play method where regret is a term representing how much better the agent would do if they played a different option [NAP+19]. Counterfactual refers to looking back over previous decisions, total regret is minimised over all decisions. The approach is intuitive and was used to make a Texas Holdem agent called Deepstack that defeated professional poker players in a study with over 44000 hands [NAP+19]. This approach would work well in “Oh Hell!” and could be considered for a future agent.

ISMCTS was used by Whitehouse et al. [WCPR13] along with knowledge-based methods to create an agent that could play the card game Spades. Knowledge-based methods refer to optimisations that stem from someone's knowledge of the game, such as the heuristics used in my informed agent. The agent created to play Spades was analysed over 27592 games that were played against real players on a mobile platform [NAP+19] to gauge its difficulty. An approach like this would be ideal for "Oh Hell!", this would allow my agents to be combined to make a stronger agent, also the user evaluation would be easier if I produced my software as a mobile game and advertised it to the public to collect a large amount of data from players with varying skill levels.

Chapter 3

Design

3.1 Making the Game

3.1.1 Classes and parameters

When designing the structure of ”Oh Hell!”, the main goal was to ensure it could be evaluated easily and quickly over many samples. This included making an efficient system that took in the different parameters of the game and output the scores of the players. This approach only allowed agents to play against each other and not for players to interact with the agents; which was adequate for early evaluation of agents' scores, for comparisons between agents, and very useful for improving the informed agent through self-play. The game is coded in Python and utilises an object-oriented programming style so that a game can be run in a single line by making a game object and can be analysed by extracting the relevant information from the object.

The game object takes in numerous important parameters which determine the attributes of the game, these are listed along with a thorough description:

- The number of players which can be set between 2 and 7.
- The player's strengths list that is the same size as the number of players, each taking an integer from 0 to 2 representing the difficulty of the agents; 0 being the random agent, 1 the informed agent, and 2 the Monte Carlo agent.
- The number of rounds parameter dictates how many times the cards are dealt.
- The hand size parameter determines how many cards are dealt to each hand in the first round

- The dynamic hand flag which, when true, decreases the size of the hand every round by 1 until the hand size is 1, after this the hand size increases every round up until the maximum hand size, where after it decreases. This repeats until all rounds are finished.

When making an instance of the game, the constructor function initialises the players, these player objects are where the important output information is stored, such as the score and the number of bids made.

The classes diagram is shown in figure 3.1, here is an overview of the classes:

- The card class simply contains two integers, one being the value of the card (a number from 2-14 where 14 is the ace since it is the highest valued card), and the other being the suit (integers from 0 to 3 representing hearts, diamonds, spades, and clubs respectively).

- The hand class can be empty or made up of cards that are stored in the class's card list. The hand class's main role is to belong to the player but is constructed by the deck in the game when dealing.

- The deck class describes an object that contains all the cards at the beginning of the game and has a shuffle function which is called at the start of each round. This class also contains the make-hand function which takes a specified number of cards out of the deck and forms hands to give to the players.

- The basic player class is the parent of the agents that follow in the following sections of this chapter. The agent is given a set of choices from the game and returns a choice randomly. The player class must hold all the relevant information about the outcome of the game since I want to extract statistics from each player. With this in mind, I gave the player score and bids-made variables. To get the player to interact with the game, I wanted to encapsulate the player's decision-making by utilising a distributed paradigm [Weg90]. This is where the game calls the player's play-bid and play-option functions when they are required to respond with a move. This allows for the game logic to be separated from the agent's decision functions.

- The game class then simply outlines the stages necessary to complete a specified number of rounds, this is detailed in figure 3.2. The "round" function in the game class takes players through the bidding phase and then receives the cards each player chooses until there are no cards left. The ”trick” function takes the cards played and determines which player won the trick, it returns the index of this player so that their score can be updated and they start the next trick.

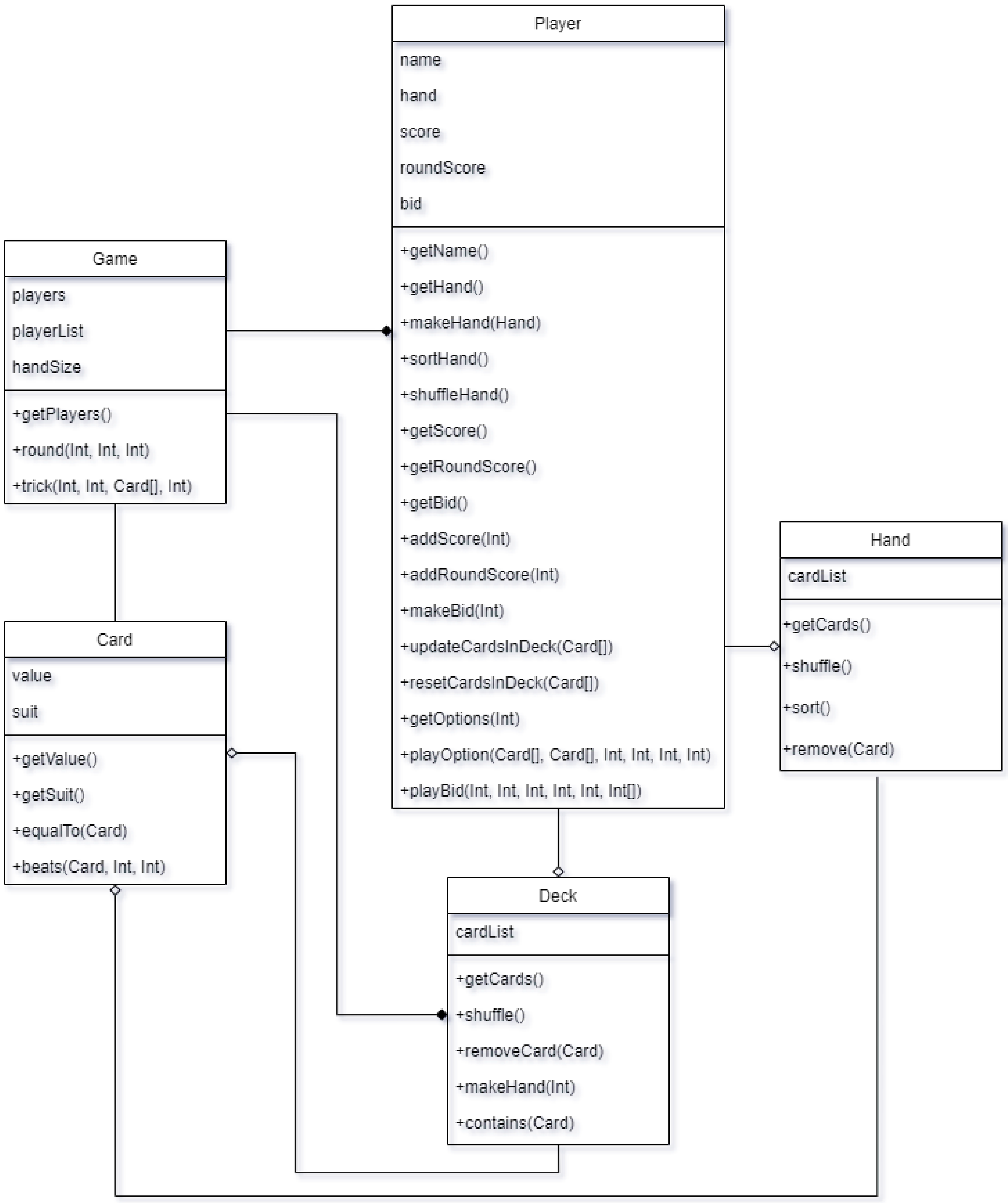


Figure 3.1: Class diagram of ”Oh Hell!” (excluding the other agents)

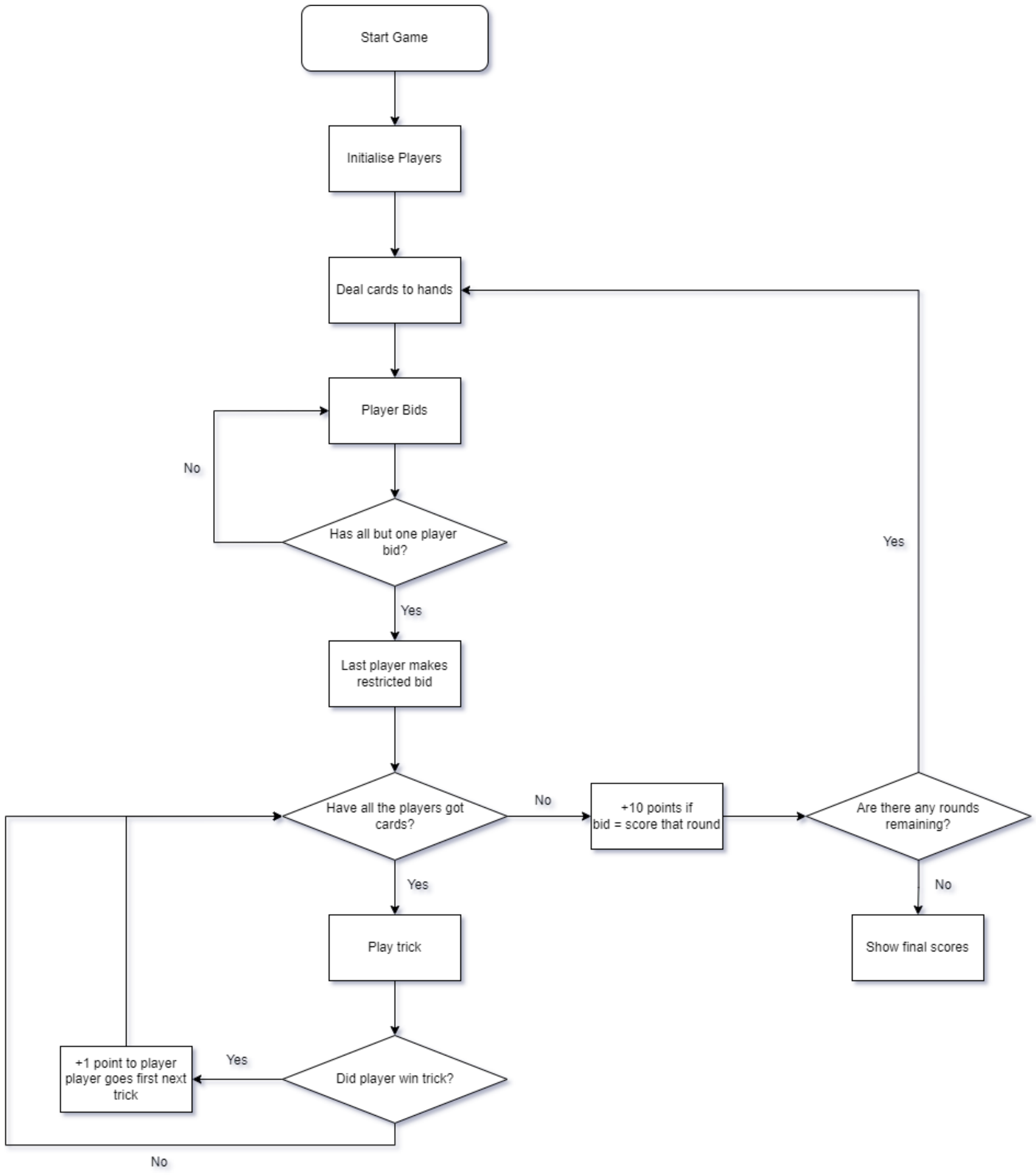

Start Game
Initialise Players
Deal cards to hands
Player Bids
No
Has all but one player bid?
Yes
Last player makes restricted bid
Have all the players got cards?
No
+10 points if bid = score that round
Are there any rounds remaining?
Yes
No
Show final scores
Yes
Play trick
+1 point to player player goes first next trick
Yes
Did player win trick?
No

Figure 3.2: Flow diagram showing the phases of the game which the game class outlines

3.1.2 User interface and interaction

The above approach allows me to simulate games quickly and is intended for evaluating the performance of the agents and testing their correctness. This does not however allow for any user interaction which is pivotal for proving a high level of performance against real players. For users to interact with the system, an interface is required. I decided to use tkinter [Lun99] to construct a graphical user interface because I was familiar with the library and believed it had all the necessary elements to implement my simplistic design shown below in figure 3.3. I used a couple of attractive assets for the interface, the first of which is a grassy background [txt15], and the other asset is a collection of sprites for the playing cards [Wol21]. Cards in the player's hand are located at the bottom of the screen; when it's the player's turn, the cards that the player is allowed to play are highlighted. The labels of information I decided to add to the interface are the key pieces of information players need to know during the game: the trump suit, the player whose turn it is, each player's bid, each player's overall and current score, the cards that have been played in the trick so far, and finally the player that won the trick and the card that won it. Additional indicators like the order of players and the person who leads the trick are implied so I did not deem them necessary to include in the interface.

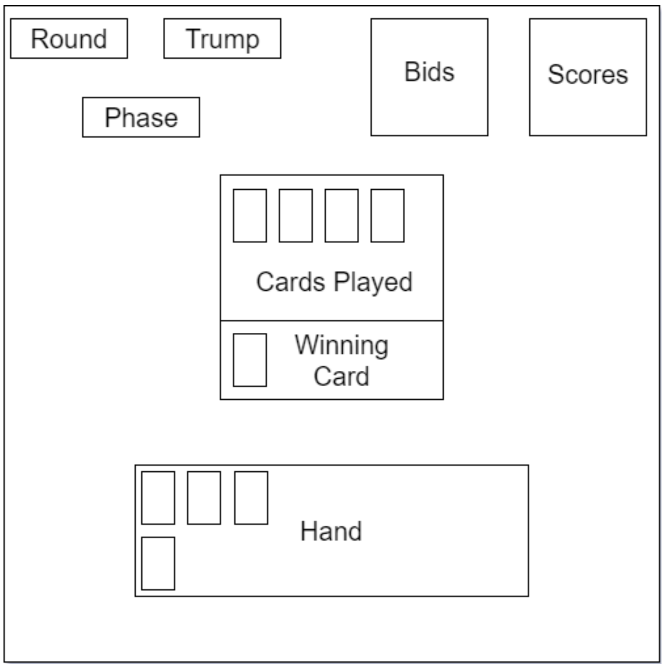


Figure 3.3: Graphical user interface design

The GUI functions similarly to the game class, the main difference is the GUI having to wait to collect the user's choices before carrying on with the game. I decided to make the most important parameters adjustable in the menu screen before starting a new game. The most important parameters are the number of rounds, the starting hand size, the number of players, and the difficulty of the agents. Once a game is finished, the scores are saved to a csv file, this feature is for user evaluation.

3.2 Informed Agent

3.2.1 The primary challenges in ”Oh Hell!”

With ”Oh Hell!” being a game of imperfect information and chance, it makes the bidding process for the agents very difficult. Predicting exactly how a game plays out is almost impossible since there is a large number of paths the game can follow. As well as this, even if you are confident in your bid, you cannot be certain that you will reach it because something unexpected can happen at any point: e.g. a trick you wanted to win gets swiped away from you by someone playing a trump, leaving you with no way to score more before the end of the round; or you did not expect to win a round with a throw-away card and now you are over bid. The only way to combat this is to know the cards in the other player’s hands.

With these challenges in mind, the informed agent attempts to take a heuristic approach to the problem by implementing well-known strategies in the agent’s decision-making function. This agent is designed as multiple different versions, each improving after the last. Each design iteration first improves the algorithm used to decide the bid, and then the algorithm that decides what card to play at a given point in the game. The next 3 sections go over each iteration of the design and give an understanding of the evolution of both the bidding and playing algorithms.

3.2.2 Informed agent version 1 - naive strategy

This agent inherits the previously mentioned player class. The first version takes a naive approach to the bidding problem, the naive algorithm is shown in figure 3.4 below in the form of a flow diagram. “bid = ban” refers to the integer that cannot be bid because of the restriction rule. This algorithm attempts to find a good bid by picking a random number between 7 and 14 for each card in the player’s hand, if this number is less than the value of the card then the bid is incremented. Any trump cards greater than 7 also increment the bid. By doing this, the algorithm is putting more emphasis on higher-valued cards being more likely to win with a relative probability scaling with their value. This is an improvement from bidding at random but the main problem with this algorithm is that the agent will often bid higher than it should since most trump cards will not win every time. The algorithm is also very random, this randomness was originally designed to combat the variance in games but can randomly make the bid more inaccurate.

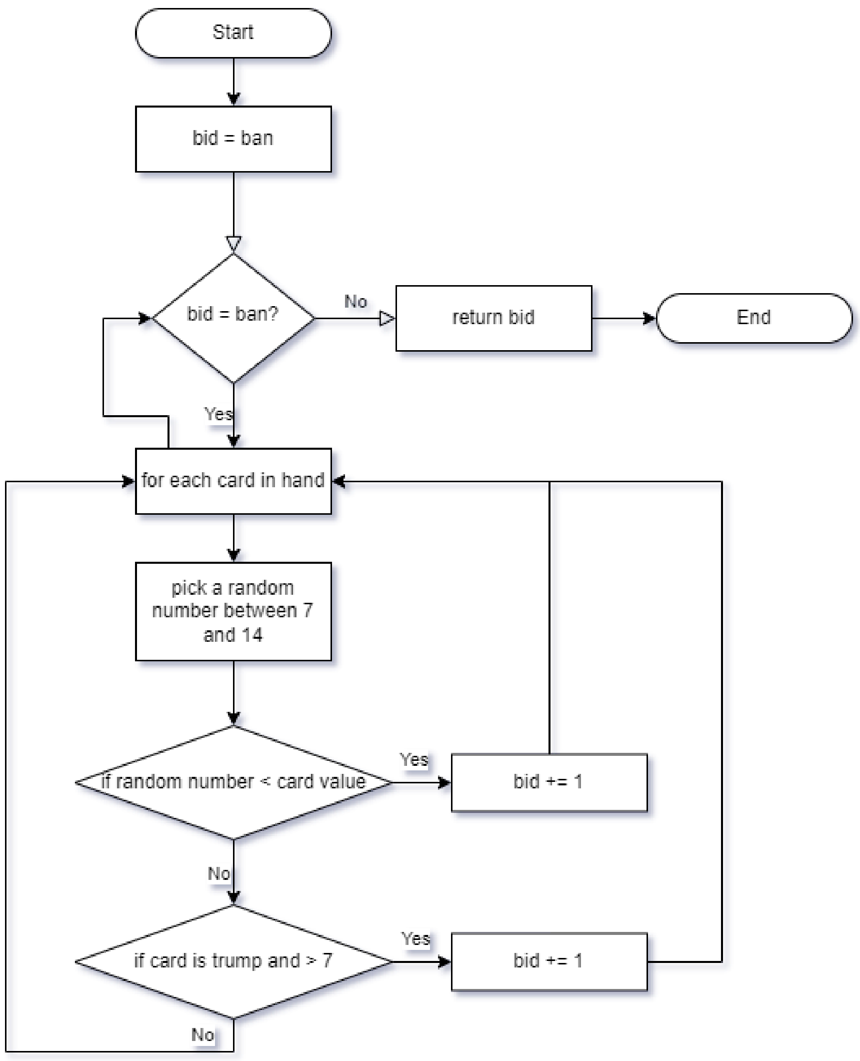


Figure 3.4: Flow chart of a naive approach to the bidding problem

The naive algorithm for choosing a card from a list of options is much more intuitive and not as random as the naive bidding. A flow diagram portrays this algorithm in figure 3.5. It starts by checking if there is only one option to pick from and returns it, this can happen often since players are forced to follow suit. After this, it checks if the player is leading the trick; if this is the case then there are no cards that have already been played, and therefore no way to know if you can win or lose the trick. To solve this I decided to simply make the agent play the highest-scoring card if the round's score does not equal the bid, and the lowest-scoring card if the round's score equals the bid. This way the player increases their odds of winning and losing when necessary to maximise their score. After this, we move on to playing when not leading. When a player does not lead the trick, there are always some cards that have been played. The last player in a trick has no uncertainty in their play because another player cannot come along and ruin it for them. With this in mind, the algorithm makes use of this by creating a subset of winning and losing options from the list of options, these are made using the algorithm from the pseudocode in algorithm 1 below. It then tries to win if the round's score does not equal the bid and tries to lose if the round's score equals the bid as before. If it cannot win and lose when it wants, then it chooses a random move instead.

Often guaranteeing a win is quite difficult when playing between first and last in a trick since following players can beat the card that players believed would win. Losing however is much more guaranteed in the way that going between first and last and having a losing option means that there exist no other cards that allow you to win because a card already beats you. However, this does not mean that losing is an easy strategy. Players are often caught out and forced to win. As well as this, if you are always bidding lower than your opponent, and both you and your opponent are making an equal number of bids, then the difference between you will be the points per trick. In general, I think the naive first version reflects a novice-level strategy, which is a foundation for developing the heuristic-based agent further.

Algorithm 1 Finding options that win and options that lose based on the cards played

```
Input: Array $options$ of size $n$, Array $cards\_played$ of size $m$
Output: Array of cards that win, Array of cards that lose

// Set first card in $cards\_played$ to winning card so far
$winning\_card \leftarrow cards\_played[0]$

// Iterate through $cards\_played$
for $card$ in $cards\_played[1:]$ do
    if $card$ beats $winning\_card$ then
        $winning\_card \leftarrow card$
    end if
end for

$winning\_options \leftarrow []$
$losing\_options \leftarrow []$

// Iterate through $options$ to see which beat the best card
for $card$ in $options$ do
    if $card$ beats $winning\_card$ then
        $winning\_options.append(card)$
    else
        $losing\_options.append(card)$
    end if
end for

Return $winning\_options, losing\_options$
```

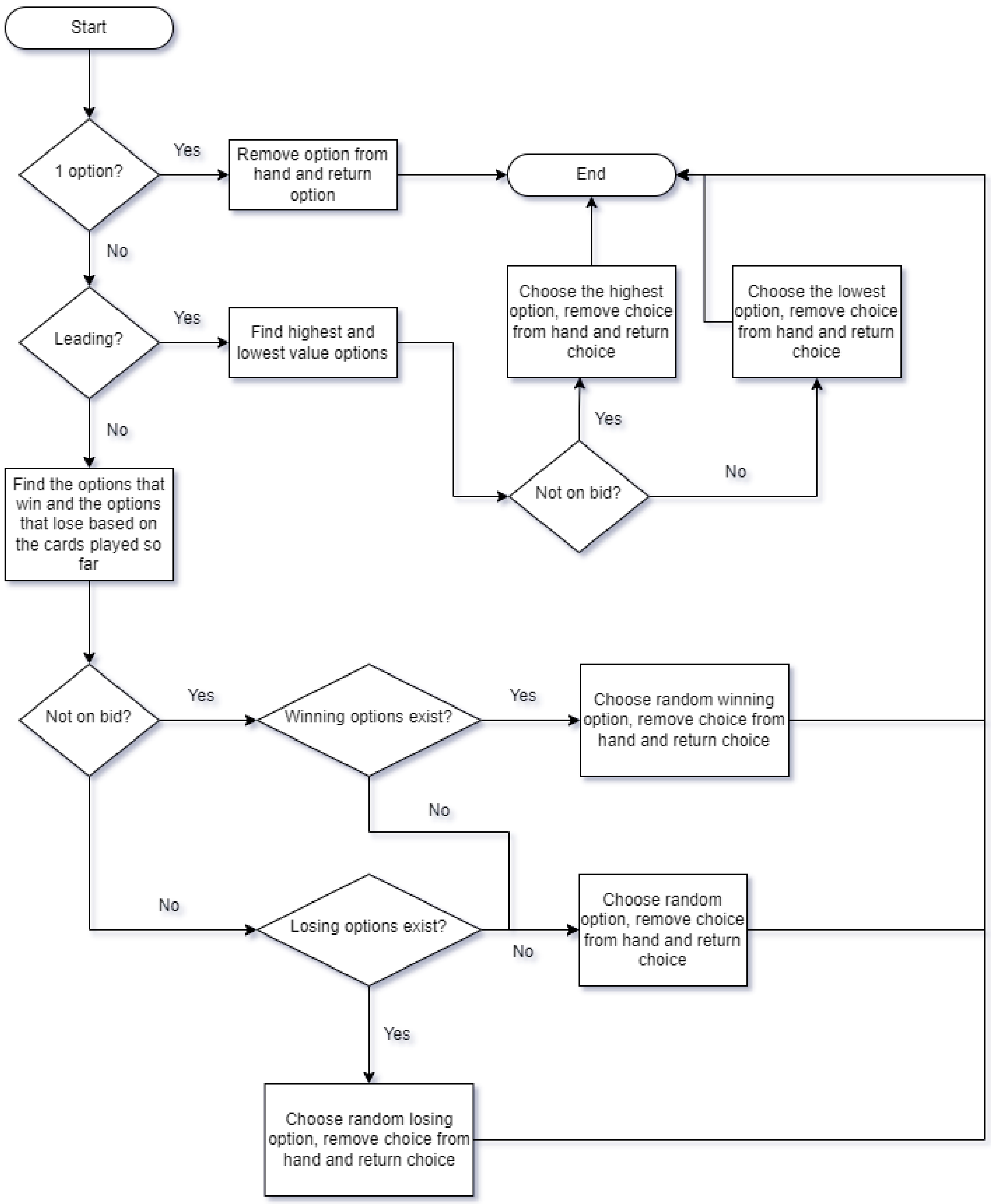

Start
1 option?
Yes
Remove option from hand and return option
End
No
Leading?
Yes
Find highest and lowest value options
Choose the highest option, remove choice from hand and return choice
Choose the lowest option, remove choice from hand and return choice
Yes
Not on bid?
No
No
Find the options that win and the options that lose based on the cards played so far
Not on bid?
Yes
Winning options exist?
Yes
Choose random winning option, remove choice from hand and return choice
No
No
Losing options exist?
No
Choose random option, remove choice from hand and return choice
Yes
Choose random losing option, remove choice from hand and return choice

Figure 3.5: Flow chart of a naive approach to the playing to bid problem

3.2.3 Informed agent version 2 - counting cards

The next version of the informed agent utilises counting cards. This is where the agent keeps track of which cards have been played and which cards could still be in other player's hands. I call this the "winning probability" heuristic where the agent calculates the probability that a card wins a trick given the cards that are still in play. The general idea is conveyed by the diagram in figure 3.6 using a much smaller sized list of unseen cards than reality for simplicity. When counting cards in this example, let $c \in C$ where c is the card whose winning probability is being calculated, and C is all the 52 playing cards in the deck. If U are the unseen cards where $U \subset C$ the winning probability of c is given by:

$$winning\_prob(c) = \frac{\sum_{u \in U} c.beats(u)}{|U|}$$

In this equation, "beats" refers to the function which is equal to 1 if card c wins in a trick over card u given the context of the round (the trump suit) and is equal to 0 if this is not the case. This can be seen as a heuristic for evaluating the value of each card in the agent's hand. Another heuristic introduced in this version is the "same suit" heuristic which is very similar to winning probability but instead of over all the unseen cards, it is only the unseen cards in the same suit, this is also conveyed in figure 3.7. In this version, the same-suit is only used when the agent is bidding or leading and is multiplied by the winning probability to make use of the fact that most cards will be played in tricks which contain cards all of the same suit. This calculation is given by the equation:

$$same\_suit(c) = \frac{\sum_{u \in U'} c.beats(u)}{|U'|}$$

This equation is very similar to the previous heuristic, but instead considers only the cards in $U' \in U$ which represent the cards in U with the same suit as card c.

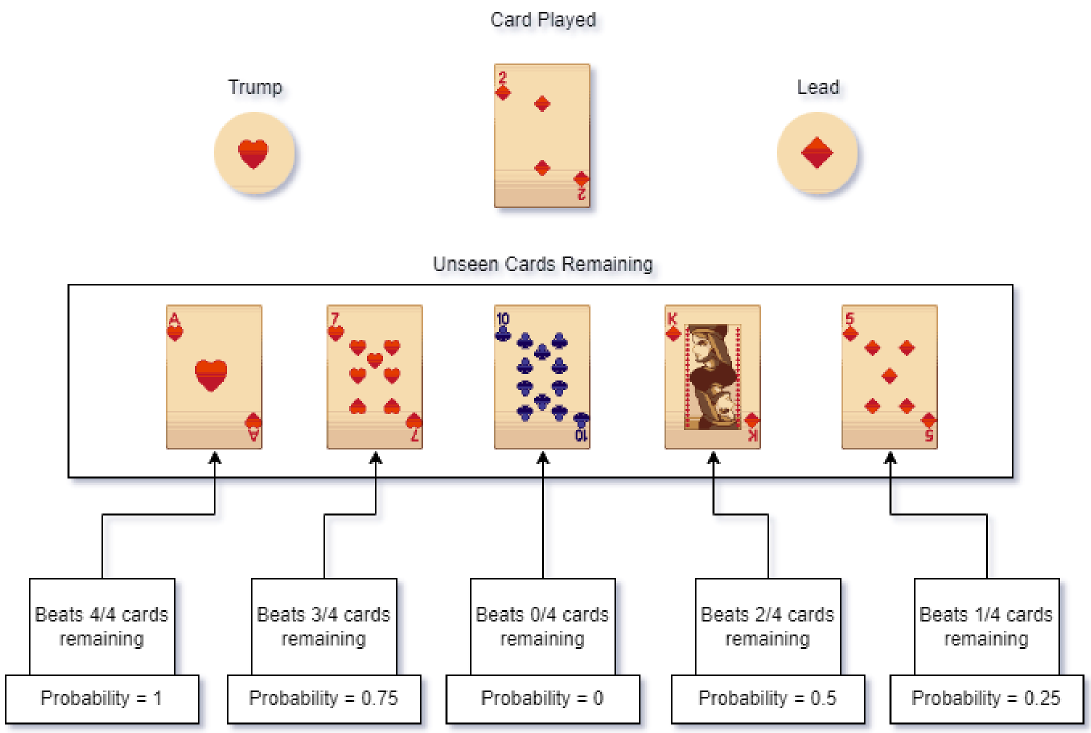

Card Played
Trump
Lead
Unseen Cards Remaining
Beats 4/4 cards remaining
Probability = 1
Beats 3/4 cards remaining
Probability = 0.75
Beats 0/4 cards remaining
Probability = 0
Beats 2/4 cards remaining
Probability = 0.5
Beats 1/4 cards remaining
Probability = 0.25

Figure 3.6: Diagram of an example of the winning probability heuristic

The bidding algorithm calculates the winning probability multiplied by the same suit probability of each card, if the card's winning probability is over a certain threshold then the bid is incremented. The threshold was manually tweaked until I believed the bid was a good reflection of the hand. Then the algorithm adds an extra bid when the final calculated bid is equal to the ban. This algorithm is a much greater improvement from naive bidding as it lacks the random element, opting for a more calculated approach. The main drawback is that it tends to overbid trump cards since trumps are favoured highly in the winning probability calculation.

Unlike the bidding algorithm, the new playing algorithm shares aspects with the naive playing algorithm. When leading, and the player wants to win more, instead of using the highest-valued card, it uses the card with the highest winning probability. The player also uses the lowest winning probability when they are on their bid. This slight optimisation makes a significant difference since the highest-value card may not have the highest winning probability. For example, the highest value card may be a king of hearts, but since we have been counting cards, we know that someone has the ace of hearts and so will be beaten.

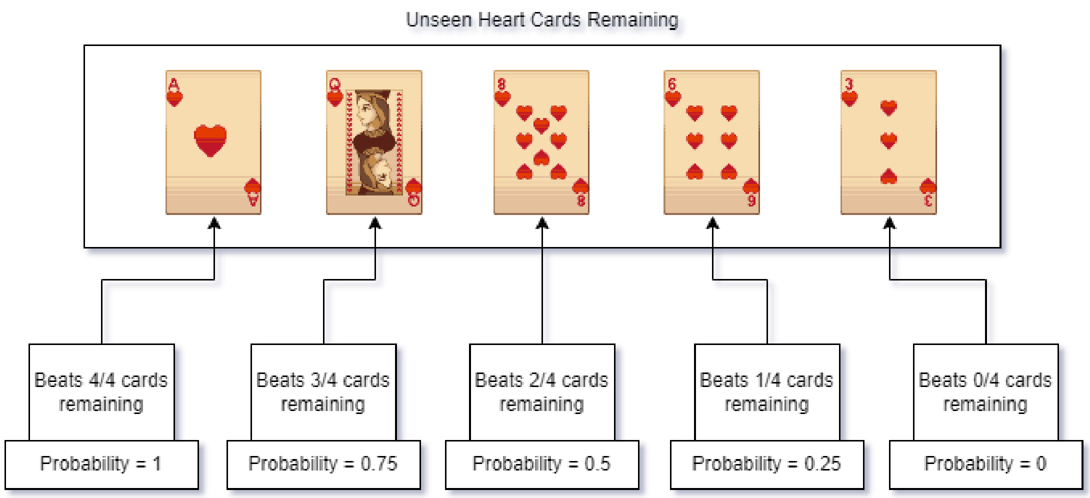

Unseen Heart Cards Remaining
Beats 4/4 cards remaining
Probability = 1
Beats 3/4 cards remaining
Probability = 0.75
Beats 2/4 cards remaining
Probability = 0.5
Beats 1/4 cards remaining
Probability = 0.25
Beats 0/4 cards remaining
Probability = 0

Figure 3.7: Diagram of an example of the same suit heuristic

The optimisation is even more impactful when the player is not leading. For instance, if the agent is trying to win since it is not on its bid, it will choose the card from the winning options with the best winning probability rather than choosing a random card from the winning options like in the naive version. This means the card is less likely to be beaten by the next player in the trick. Also, if the player is at the end of the trick and has a winning option, it will pick the smallest one since it knows it is a guaranteed win, this means it can save the card with a higher winning probability for a future win. The algorithm does a similar procedure when it wants to lose, making sure to get rid of high cards when it is safe so that it is less likely to be forced to win. The agent still chooses a random card when it is forced to win or lose when it does not want to.
This design is a significant upgrade from version 1 due to the playing algorithm using a much more human-like strategy. From experience, real players will not throw away their cards randomly when posed with the option, instead they have the foresight to save them for future tricks.

3.2.4 Informed agent version 3 - a weighted combination of multiple heuristics

The final version upgrades the evaluation of cards. The algorithms for bidding and playing a card are the same as version 2, the difference between these agents comes from balancing the weighted combination of five different heuristics to give a more precise value to each card. The five heuristics include the two heuristics used in the previous version as well as three new heuristics. The first of which is called ”follow-suit” and is roughly outlined in figure 3.8. The general idea of this heuristic is to give the likelihood that other players play a card that is the same suit as the card being evaluated. The motivation for utilising this heuristic comes from a strategy called ”voiding” [Par08] existing in most whist variations. This is when you try and lose all cards of a suit in your hand in the hopes that when another player plays a card of that suit, you can play any card in your hand freely without being restricted by following suit. When follow-suit is high for a card, there will be a higher proportion of cards of that suit in the list of unseen cards, meaning less of them in the agent’s hand as well as a higher chance of them being played. Therefore, the follow-suit heuristic increases as the chance of being able to utilise the voiding strategy increases.

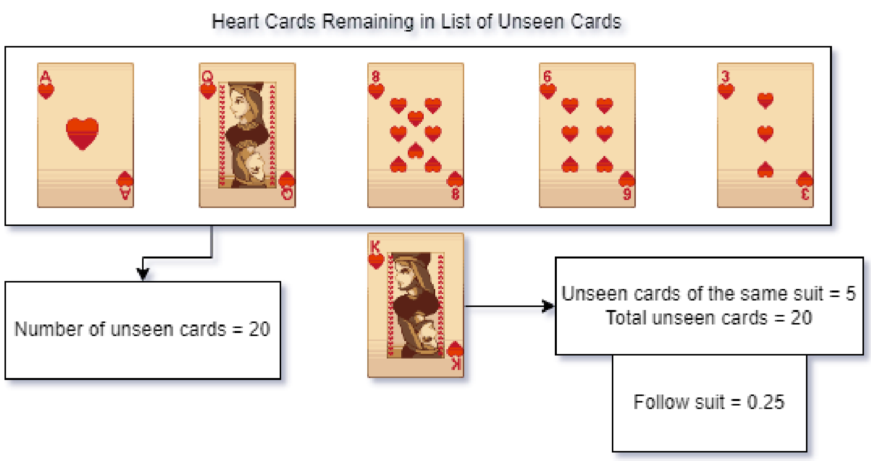

Heart Cards Remaining in List of Unseen Cards
Number of unseen cards = 20
Unseen cards of the same suit = 5
Total unseen cards = 20
Follow suit = 0.25

Figure 3.8: Diagram of an example of the follow-suit heuristic

This version also uses the ”trump-probability” heuristic. As the name suggests, it gives us the likelihood that the card being evaluated gets a trump card played on it. This can be seen as the defence to the voiding strategy, where the agent wants to know the probability that a card they play can be beaten by a trump card. An example of the calculation can be seen in figure 3.9.

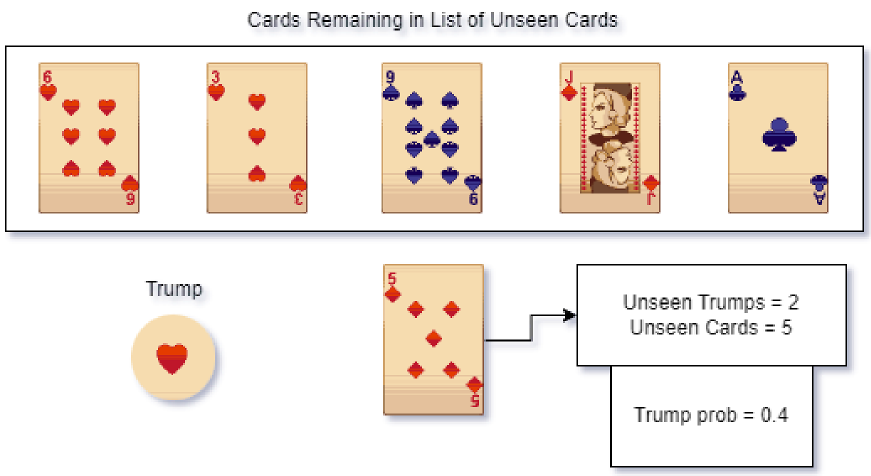


Figure 3.9: Diagram of an example of the trump-probability heuristic

The final new heuristic is called ”suit-safety”. The goal of this heuristic is to make sure the agent is not forced to play a card they do not want to. An example of this could be that the agent has a queen of hearts which the agent may want to use to win a trick since its value is high. However, if the agent’s only heart was this queen, any other player could lead a trick with the king of hearts, forcing the agent to play the queen of hearts and lose the trick which could cause the agent to not make their bid. The general idea of suit-safety is that if the agent has more cards of the same suit, the probability that they will be caught out like in the above example would be lower. This is illustrated in figure 3.10 below.

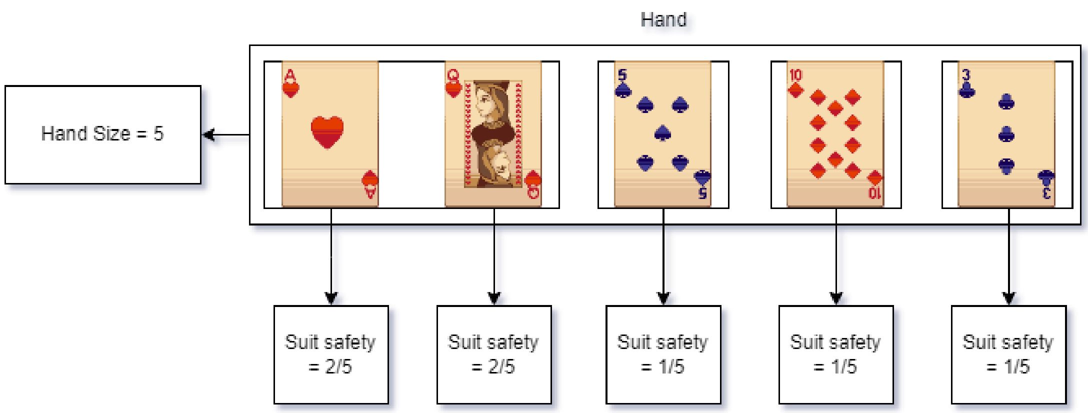


Figure 3.10: Diagram of an example of the suit-safety heuristic

In the evaluation of the cards, three different scenarios have to be considered. The first is when the player is bidding. In this stage, I believe that the winning probability does not properly reflect the value of the cards, since trump cards beat all other cards, and in a real game, the lowest trump will often be baited out and so is not worth much at all. Removing this heuristic avoids the problem of overbidding when the agent's hand contains numerous trumps. Instead, I used the same suit heuristic as well as the three new heuristics which I believe are all relevant to estimating the bid. The next stage that uses card evaluation is when the agent is leading a trick. This stage uses a combination of all five heuristics as I believe they are all relevant when the agent is playing first in a trick. The last scenario to consider is when the agent is playing a card on a trick where cards have already been played. I believe that the suit-safety heuristic is not necessary for this scenario since the main problem it wishes to combat that is described when introducing the heuristic is not present. With each separate scenario, the optimal weighting of each heuristic will differ between situations, therefore the total number of weights that must be optimised totals 13.

The process of optimising these weights can then be performed through self-play. Firstly the agents are given a list of arbitrary weights which are normalised for each scenario since the game requires the card evaluation to equal between 0 and 1. From this starting point, the Optuna library is used to create a study with 200 trials. Over each trial, Optuna [ASY$^+$19a] changes the weights for the first agent (player 1) to maximise an objective function. The objective function runs 200 games to try and get a good estimation of the performance of the agent, the average score of player 1 over these games yields the output of the objective function that is to be maximised. During the trials, if the objective function returns a higher average score than the previous highest score, the weights of the opposing agents are set to the new best weights of player 1 from that trial. This process allows the average score of the informed agent to iteratively improve over the 200 trials.

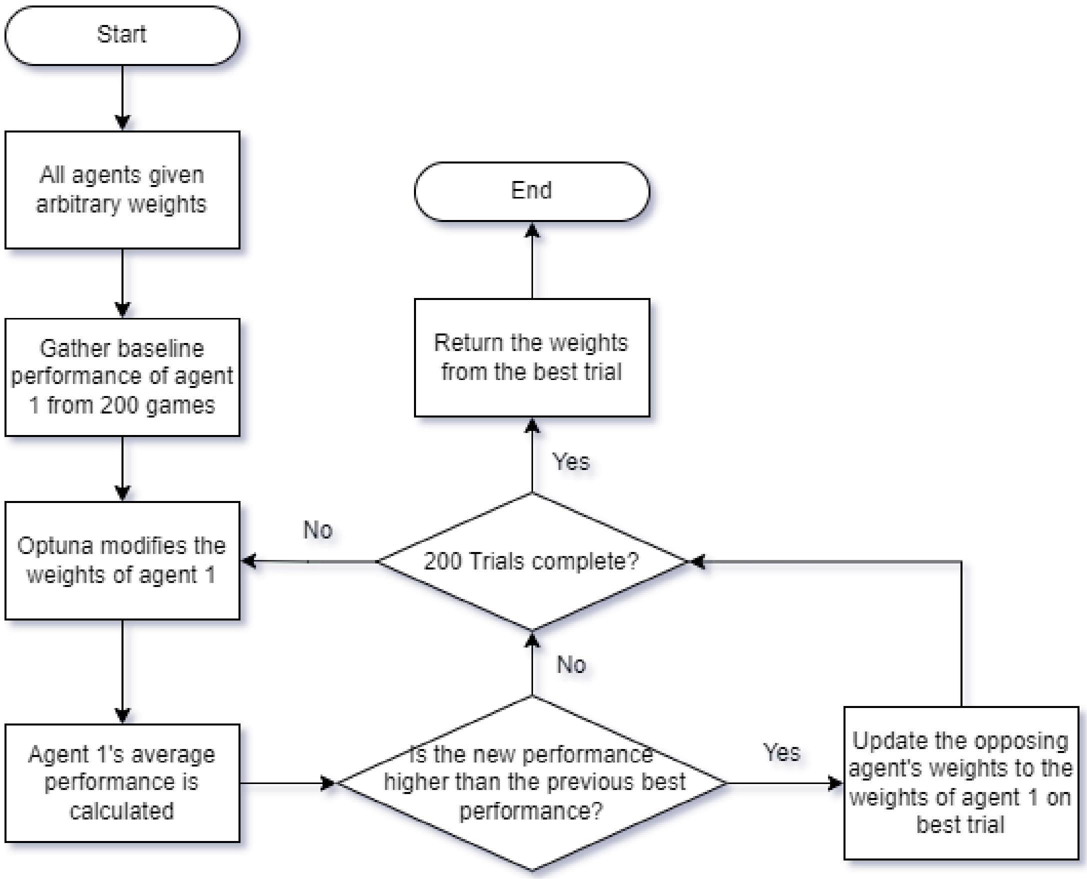

Start
All agents given arbitrary weights
Gather baseline performance of agent 1 from 200 games
Optuna modifies the weights of agent 1
Agent 1's average performance is calculated
Is the new performance higher than the previous best performance?
Yes
Update the opposing agent's weights to the weights of agent 1 on best trial
No
200 Trials complete?
No
Yes
Return the weights from the best trial
End

Figure 3.11: Flow diagram of the weight optimisation algorithm using Optuna

3.3 Monte Carlo Agent

3.3.1 The motivation behind Monte Carlo Tree Search (MCTS)

Though the informed agent makes use of good strategies, there could be more optimal strategies that have not been discovered. The informed agent also does not plan ahead and only plays in the moment without thinking about the future of the round. This is where MCTS can help; by assessing the value of different future game states with random playouts, the MCTS algorithm can provide an incredibly accurate evaluation of different choices. Due to the nature of random sampling, the MCTS algorithm converges to the game-theoretic optimum given enough time [BPW$^+$12]. This does not require any heuristics for the evaluation of a card, the evaluation is purely done through random sampling. Random playouts are the default policy for the MCTS algorithm. The tree policy I used was the UCT algorithm:

$$\text{UCT} = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

Where the game state with the highest UCT value is selected to be expanded. $\bar{X}_j$ refers to the average wins (wins/visits) of the game state j and is called the exploitation term since it favours nodes with a high value that have already been selected. n_j is the number of visits to j, and n is the number of visits to the parent of j. C_p is a constant usually called the exploration weight, this dictates how often the UCT algorithm selects less visited nodes [BPW$^+$12]. Further motivation behind the use of MCTS is that I was already well-versed in the algorithm and knew what it was capable of. I did not know however of the challenges of applying MCTS to a game like ”Oh Hell!”.

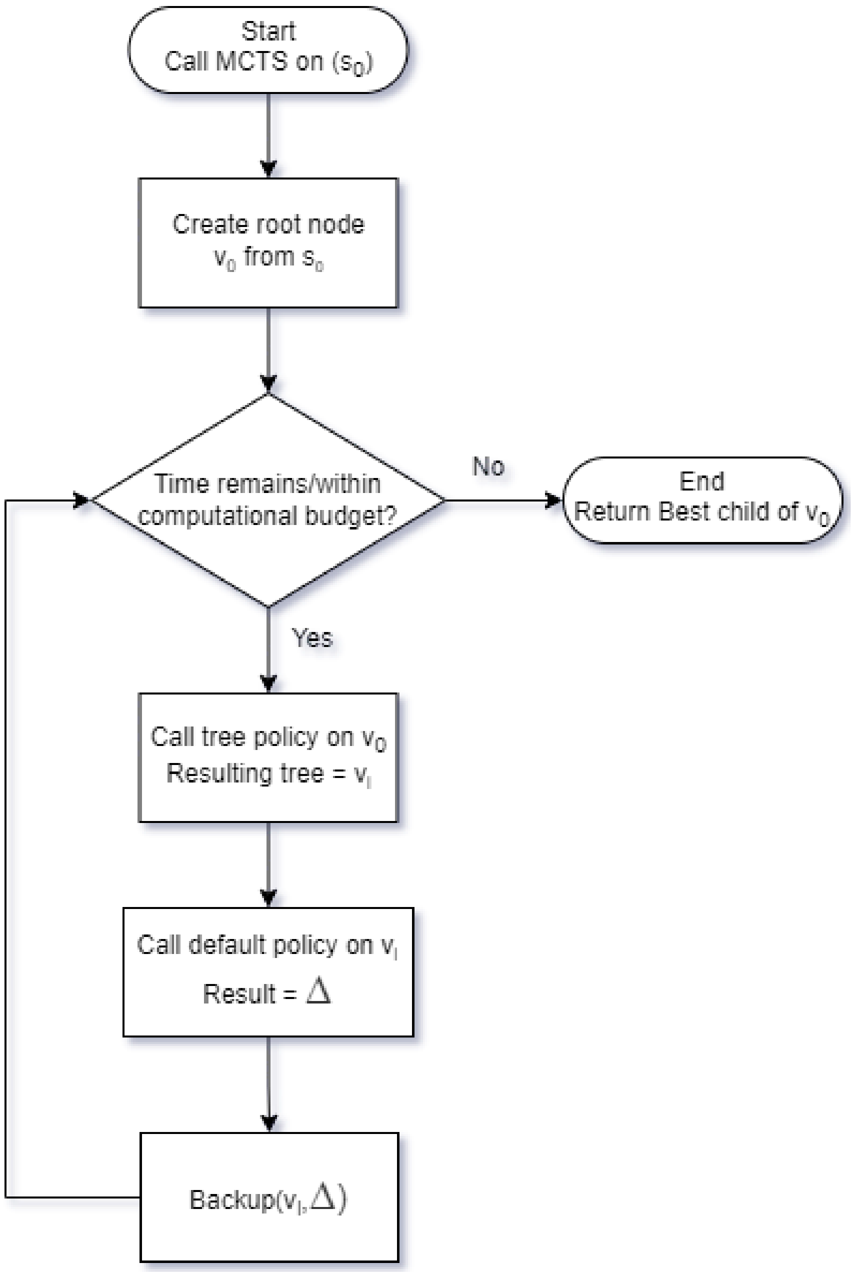

Start
Call MCTS on (s_0)
Create root node v_0 from s_0
Time remains/within computational budget?
No
End
Return Best child of v_0
Yes
Call tree policy on v_0
Resulting tree = v_l
Call default policy on v_l
Result = Δ
Backup(v_l,Δ)

Figure 3.12: Flow diagram outlining the phases of the MCTS algorithm

3.3.2 The challenges of applying MCTS to ”Oh Hell!”

”Oh Hell!” is a 3 to 7-player game with imperfect information, the most common application of MCTS that I had seen previous to this project was on games with 2 players with perfect information such as Chess or Go [HWCH18]. Games of perfect information are those where there is no uncertainty, and the winner is determined purely by the player’s decisions and strategy. These are games where the player always knows the precise game state they are in; they know all the moves they can make as well as all the moves their opponent can make in response. As mentioned in the background chapter, this type of game is unlike ”Oh Hell!” which is a game of imperfect information. The player does not know the game state they are in because they do not know what cards the opponents have, therefore they do not know how the opponent will respond.
To solve this issue, I used the previously mentioned Information Set Monte Carlo Tree Search (ISMCTS). An information set is the set of states the player could be in given their current knowledge. In ”Oh Hell!”, this would be every possible distribution of the unseen cards in the opponents’ hands, whilst the player’s hand remains the same. The ISMCTS algorithm brings a new step into MCTS called determinisation which is conveyed in figure 3.13. This adds more random sampling by repeatedly evaluating game trees by taking possible game states from the information set as the root. In the same way that infinite iterations of MCTS will converge to an optimal move, ISMCTS will do the same over every possible game state in the information set. The scores from each sample of the information set are combined at the end to give the best possible move given the agent’s current knowledge. Though it will introduce a greater trade-off in the quality of move and time taken, ISMCTS is necessary for a game tree to be built.

The other problem with building a game tree for ”Oh Hell!” is the number of players. Games like Chess and Go are 2-player games, often including a maximising player and a minimising player. I solved this by making each player maximise their own individual score. This does make players greedy, but I believe this reflects how most real players play the game anyway. Purposefully sabotaging your opponents tends to be a high-level strategy, it would mean possibly putting your own round at risk to hinder other players. The alternative to greedy agents is to have every opponent modelled as one single minimising opponent, but this would introduce the problem of loss of individuality within the opponents. Players would be modelled as working together against the agent, but in reality, the game is often very individual.

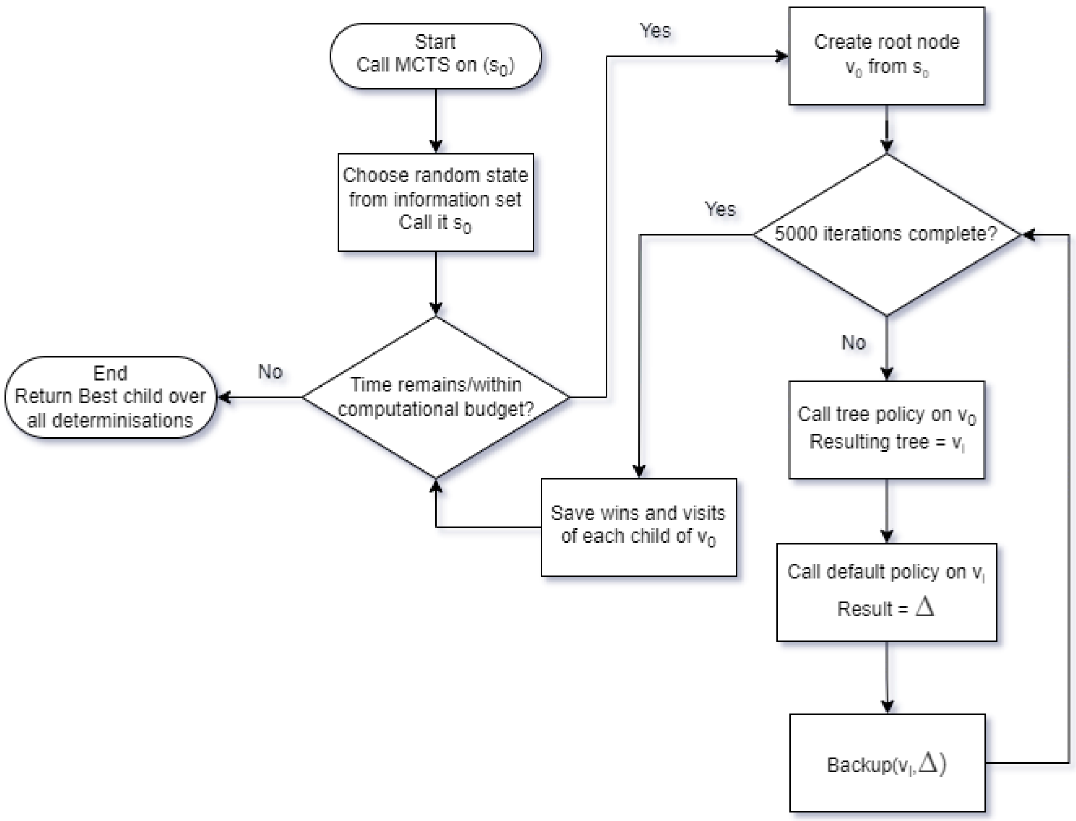

Start
Call MCTS on (s_0)
Choose random state from information set Call it s_0
Time remains/within computational budget?
No
End
Return Best child over all determinisations
Yes
Create root node v_0 from s_0
5000 iterations complete?
Yes
No
Call tree policy on v_0
Resulting tree = v_l
Call default policy on v_l
Result = Δ
Backup(v_l, Δ)
Save wins and visits of each child of v_0

Figure 3.13: Flow diagram outlining the phases of the ISMCTS algorithm

3.3.3 Designing the Monte Carlo Agent

This agent, like the informed agent, inherits the player class. The functions inside the agent are relatively straightforward, the complexity comes from the game tree class and the different functions within it. The bidding function takes in:

- The restricted bid which is only relevant if the agent is bidding last
- The size of the hand for this round
- The trump suit
- The number of players
- The other player's bids so far
- The agent's playing order

The function then follows the regular ISMCTS algorithm taking random samples of the information set within a time limit of 5 seconds. This time limit was chosen by trial and error, my criteria was making sure to balance the algorithm's acceptable (in my opinion) estimation for the bid while keeping the time taken to be as long as a typical player would take to produce their bid. Each game tree takes 5000 iterations, which was also produced through trial and error, with the goal being that the value of the best move stabilised around this number on average. After saving the wins divided by the visits of each move over each sample of the information set, the algorithm takes the average over each sample to estimate the bid. This value is rounded to the nearest integer. When the agent is restricted in its bidding, it rounds the estimated value of the hand up or down to avoid the restriction. Detailed pseudocode is given for this in algorithm 2 below. The evaluation produces the average score at the end of the round after considering the outcome of leading with each card in the agent's hand, I believe that this will produce a decent approximation for the bid because it leaves an element of random variation like before but provides a more statistical approach due to the large number of simulations. The function the agent uses to choose a move takes in the possible moves the agent can make, as well as the cards played in the trick so far, the trump suit, the number of players, the list of bids each player has made, the scores in the round so far, and the order the player is playing in. The function begins similarly to the informed agent, where if the agent only has one option, it returns it. If this is not the case, the function then proceeds with the ISMCTS algorithm. The time limit

Algorithm 2 Algorithm used by the Monte Carlo agent to compute a bid

```
Input: Integer $ban$, Integer $hand\_size$, Integer $trump$, Integer $players$, Array $bids$ of size $players$, Integer $current\_player$, Array $unseen\_cards$
Output: Integer $bid$

// Initialise variables for limiting iterations and for counting wins
$time\_limit \leftarrow 5$
$iterations \leftarrow 5000$
$wins \leftarrow []$ of size $hand\_size$
$samples \leftarrow 0$

while $current\_time < time\_limit$ do
    $samples++$
    make $game\_tree$
    $game\_tree.determinise(unseen\_cards)$
    for $i \leftarrow 1$ to $iterations$ do
        $selection \leftarrow game\_tree.select\_child()$
        $expansion = selection.expand()$
        $simulated\_value = expansion.simulate()$
        $expansion.backpropagate(simulated\_value)$
    end for
    for $x, child$ in $enumerate(game\_tree.children)$ do
        $wins[x] += child.wins/child.visits$
    end for
end while
$bid = mean(wins)/samples$

// If bidding last
if $len(bids) == players - 1$ then
    // If over bid round down if possible
    if $sum(bids) > handSize$ then
        if $floor(bid) == ban$ then
            $bid = ceiling(bid)$
        else
            $bid = floor(bid)$
        end if
        // If under bid round up if possible
    else
        if $ceiling(bid) == ban$ then
            $bid = floor(bid)$
        else
            $bid = ceiling(bid)$
        end if
    end if
else
    $bid = round(bid)$
end if
Return $bid$
```

and iterations are set to the same as the bidding phase since I believe the time frame for making a move is still relevant. After each sample has been searched, the wins divided by visits for each move are saved, once the time limit expires, the move with the highest wins divided by visits is chosen to be played and the move is returned.

The game tree is created at the beginning of each sample of the information set. The variables required to make a state in the game tree are passed into the object as parameters when being initialised. The order in which the players are initialised is very important since it dictates at which levels of the tree are in the agent's perspective. This value is an integer from 0 to the number of players minus 1, most other parameters are passed into the constructor in a way that means the order indexes their position in the list. This holds true for scores, bids and player's hands. The original list of player's hands is empty other than the entry indexed at the agent's order which contains the agent's hand. Figure 3.14 encapsulates this graphically below in the form of an example of all the different parameters that go into a single state in the game tree.

This example shows a game state that would form the root of player 2's game tree. This game tree is created at the first trick of a round after the bidding phase has been completed. In this example, player 2 has bid 2 as seen by the second item in "each player's bid" which is within player 2's knowledge. Player 1 has already played a card, meaning the size of player 1's hand that is to be determinised is less than the other 2 players. Once this root node has been initialised, the first sample from the information set is generated, this is done by calling the determinise function. This function takes in the cards that are unseen and randomly distributes them into the hands of each player.

Once this is complete, the rest of the tree can be built. After this step, the following functions describe the regular MCTS algorithm, the first of which is the selection function. If possible moves of the agent have not yet been expanded, then this function selects one of these moves. If all moves in a node have been expanded, then the UCT algorithm [BPW$^+$12] calculates which child node should be selected next.
Once a move has been selected, the expand function is called, creating a new node proceeding from the previous game state as if that move had been played. This new node would be from the perspective of the next player. The expand function uses logic from the game to create the new node and updates any necessary values e.g. score if the trick was finished in the state, or the current player's perspective since that player

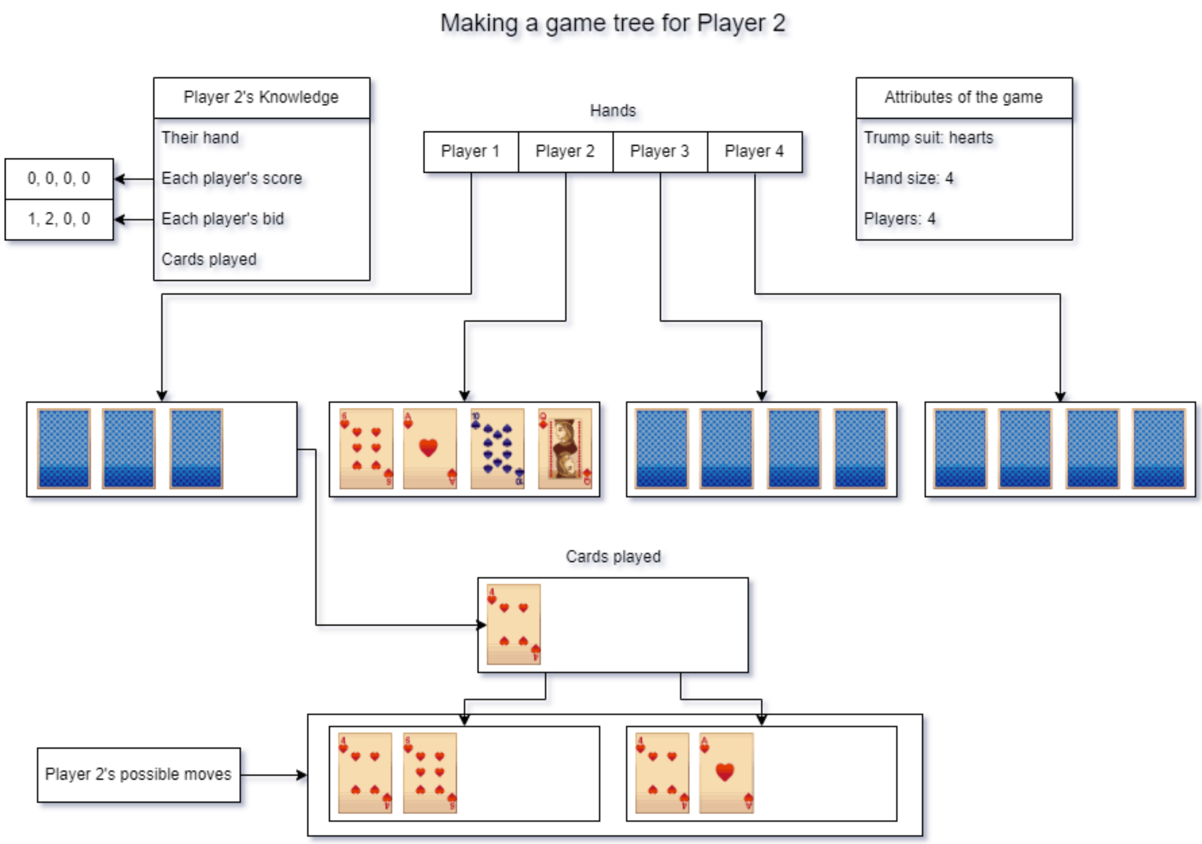


Figure 3.14: Diagram of an example of a state in the game tree

won the trick.
The simulation function then takes this node and completes a run-through of the round until termination, this is called a random playout because the moves taken to complete the run-through are random, but still follow the game logic. The scores of the players are updated as the tricks are played in the simulation. Once the end of the round is simulated, the results of the playout are returned. These results include the bonus of 10 points awarded if the player's scores match their bids.
With this list of scores, the backpropagation function updates the wins and visits of the respective players at each layer of the tree. When modelling the tree, the root node's wins and visits are not updated as no move is made here, the child nodes of the root have their wins and visits updated for the player whose move transitioned to that state. For instance, in figure 3.15 below, the wins and visits of each move are displayed. The move that the player made is credited for this. The game that made this tree had a hand size of 2 so that the simulation phase would be the same each time. This tree was generated after just 8 iterations. Once the simulation produces a list of scores at each

iteration, the wins are increased by the respective score for each player and the visits are incremented by 1.

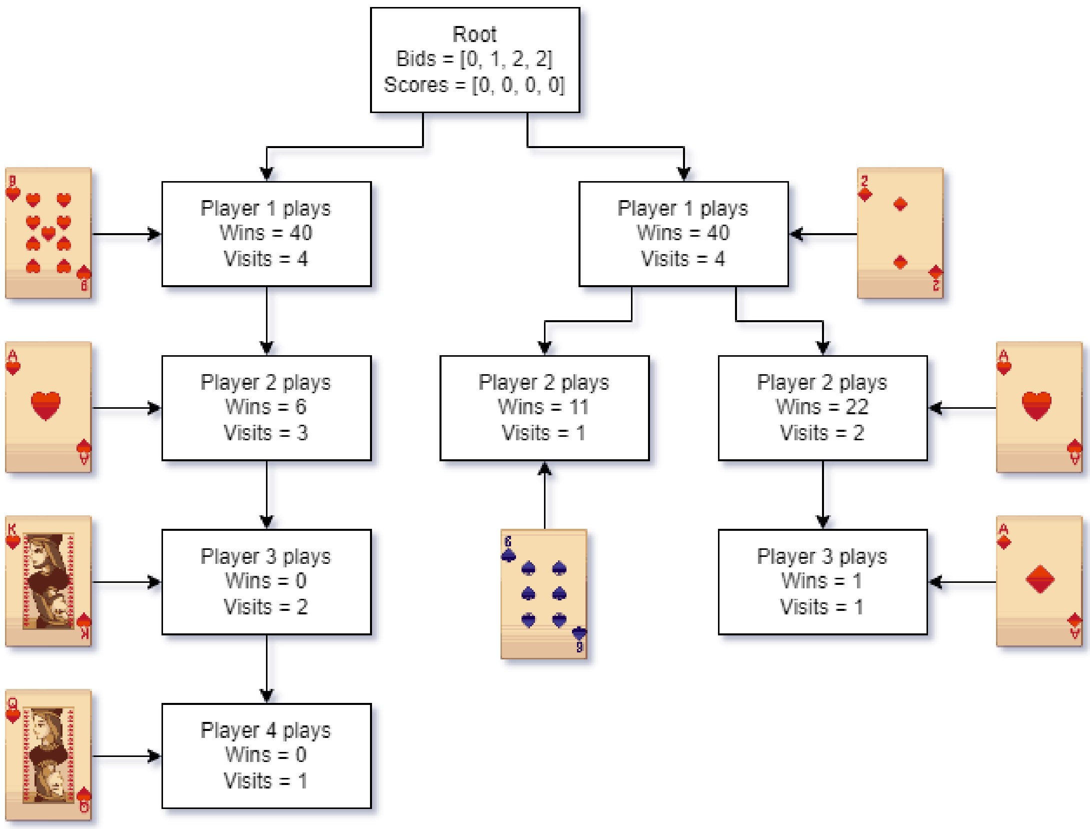


Figure 3.15: Diagram of an example of a game tree showing the wins and visits of each player and the card that yielded this

3.3.4 Optimisations

ISMCTS can often find a move quickly when given few options to choose from. For example, if the agent had a card that it favoured much more to win over most samples in the information set, the above design would unnecessarily use the full time-limit before providing its choice. To combat this, I decided to make the agent check if a card was being favoured (had the highest wins over visits ratio) for 10 determinisations. If this was the case then it would return its choice early. I believe that the chance a poor move is evaluated to be the best 10 samples in a row is very unlikely so this optimisation is fairly safe, with very little trade-off to the quality of the move.

Another optimisation I designed is similar to the previous one but is done during the construction of each game tree. I decided to return the tree once a move reached 1000 visits. This is partly so that more samples can be searched within the time-limit, and also because, at this threshold, the algorithm tends to focus on exploiting the highly visited node for the remaining iterations.

The final optimisation for MCTS is to the default policy. This describes how playouts are completed in simulation where the agent reaches a terminating state by playing randomly until they run out of cards. Default policies must be fast so random playouts are simple and adequate for quick simulations. However, if I were to improve the roll-out policy to not take random decisions, I could drastically improve the estimated value of nodes in the tree. A simple roll-out strategy that would not increase processing power but will avoid terrible moves follows a similar algorithm I used for the informed agent: if the player is on their bid, try and lose tricks. Players always want to get their bonus unless they are already overbid. This simple strategy means that simulations where the player unnecessarily goes over bid never occur. Implementing other strategies into the default policy could give a more accurate estimation but at the cost of loss of generalisation.

A further optimisation that could be made is tuning the hyper-parameters in the ISMCTS algorithm. This would be done through self-play similarly to the informed agent. The hyper-parameters include the exploration weight of the UCT algorithm, both early termination thresholds mentioned previously in this section, and the maximum depth of the game tree. The issue with doing this is that the time it takes to run games using the Monte Carlo agent is exceptionally longer than the informed agent, making a problem evaluation of the agent very difficult. With more optimisations to the speed of the agent, hyperparameter optimisation could be achieved over a long time frame.

Chapter 4

Implementation

4.1 Deciding the Winner of a Trick

This chapter will cover important snippets of code that I believe were challenging to implement and utilise notable solutions to overcome. The first of which exists in the "Game" class that was mentioned in the previous section. The code snippet below decides the winner of a trick based on the context of the game. At first, I believed that I had to go through every card and check with all the others that it was a winner, but then I realised that the cards in the game beat each other in a transitive relation as visualised in figure 4.1. This meant that I only had to iterate over each card once and save the winner so far as this would mean it would beat the others as well.

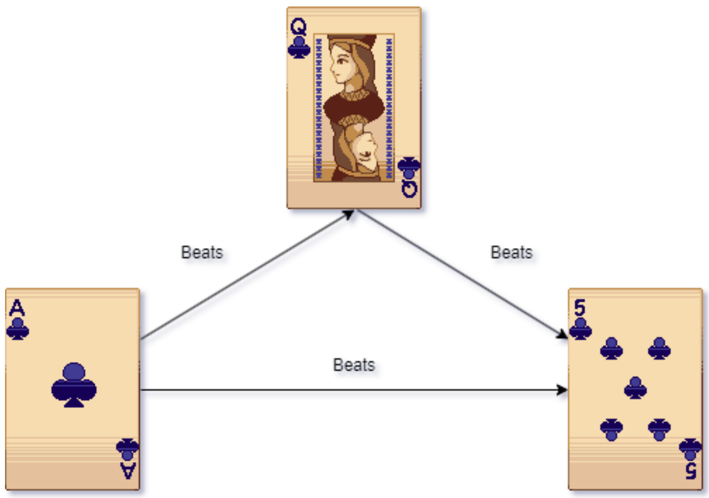


Figure 4.1: Image showing the transitive relationship between cards beating each other

Here is the code to implement this followed by a thorough description:

```python
1  def trick(self, trump, first, cardList, players):
2          # Winner is set as the first player to start
3          winner = first
4          winnerCard = cardList[0]
5          leadSuit = winnerCard.getSuit()
6          # Count needed to format player turns correctly
7          i = 1
8          for card in cardList[1:]:
9              if card.beats(winnerCard, leadSuit, trump):
10                 # Set winner to current index of card played
11                 winner = (first + i) % players
12                 winnerCard = card
13             i += 1
14         print("Card that won:")
15         print(winnerCard)
16         # Return the index of the player that won
17         return winner
```

The function takes in the context necessary for the trick to be evaluated.

- The trump suit is needed to check if a card is trumped
- The player going first is required to calculate the index of the player who won
- The card list contains the cards that each player entered into the trick
- Players represents the number of players in the game

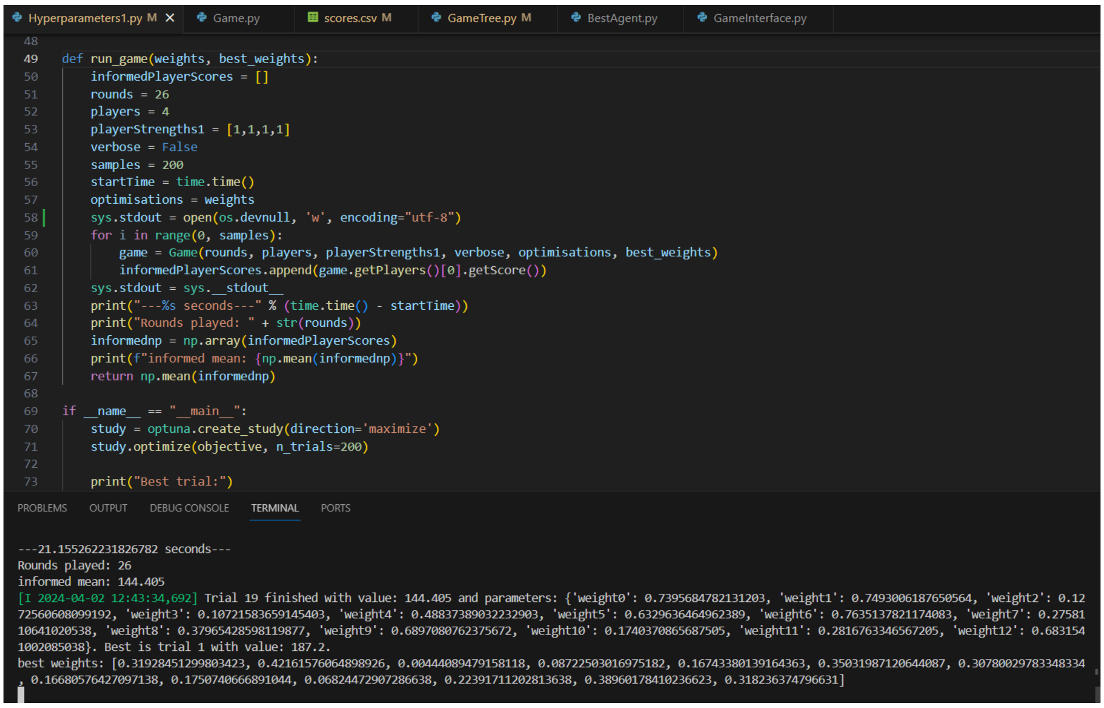
The winner is initially set to the first player, and so the winning card is the first entry in “cardList”. A counter is needed to update the index of the winning player, for example, for a 4 player game, player 3 could be going first and is indexed at 2, therefore if player 1 is the winner then the counter would equal 2 since player 1 follows after player 4. This index is then used in the calculation: $winner = (first + i) \bmod players$ so that the index stays within the number of players. The loop goes through each card excluding the first one and checks if it beats the winning card. The function on line 9 takes in the context of the trick and returns true if the “card” is a greater value than the “winnerCard” being passed into it. If this is true then the new winner is updated. This function only needs to return the index of the winner because the result of the trick only affects the score of the player who won.

4.2 Informed Player's Self-Play

In this section, I look more in-depth at the self-play algorithm and how I decided to implement it. The implementation is shown below in the code snippet, I utilised objects from the "Optuna" library which is initialised in a separate part of the code. I did not deem it necessary to show this here since the setup was created with reference to the Optuna documentation [ASY$^+$19b]. A study object is created to optimise an objective function, this is provided below.

```python
def objective(trial):
weights = []
best_weights = []
if trial.number == 0:
    # Set initial weights manually
    weights, best_weights = [0.285903516922823,
        0.1343298885883228, 0.2513171910666505,
        0.2654805206104466, 0.06296888281175703,
        0.19781578320406684, 0.40322548219933846,
        0.1057952046865951, 0.29316352990999955,
        0.24951563665051385, 0.25987660163497556,
        0.23694603672039563, 0.25366172499411493]
    # Add weights to study's first trial
    for i, weight in enumerate(weights):
        trial.suggest_categorical(f'weight_{i}', [weight])
else:
    # Find best trial to set to opponent's weights
    best_trial = study.best_trial
    for key, value in best_trial.params.items():
        best_weights.append(value)
    best_weights = normalise(best_weights)
    print(f"best weights: {best_weights}")
    # Get new suggested weights
    for i in range(13):
        weights.append(trial.suggest_float(f'weight{i}',
           0, 1))
    weights = normalise(weights)
score = run_game(weights, best_weights)
return score
```

The piece of code above is the objective function which in this case is attempting to maximise the score of the agent playing first. The weights are initially set to the weights produced from the previous 200 trials I ran. The for loop on line 8 takes the weights from the variable I passed in and adds them to the study as trial 0. Once the first trial is complete with the initial weights used as a starting point, the trials after that take the best weights as the opponent's weights. Whenever new weights are generated, they are not normalised. In this context, this means that the weights for bidding, playing first, and playing after first must each add up to 1. Since the weights are generated between 0 and 1, they must be added up over each section and divided by however many heuristics are in each part. This is what the normalise function does that appears in lines 15 and 20. Finally, once the weights have been decided, they are passed into the evaluation function named "run_game". This function simulates 200 games every time it is called to get an accurate performance estimate. The code for this function is shown in the screenshot in figure 4.2 along with the typical terminal output from running the study.



```python

def run_game(weights, best_weights):
    informedPlayerScores = []
    rounds = 26
    players = 4
    playerStrengths1 = [1,1,1,1]
    verbose = False
    samples = 200
    startTime = time.time()
    optimisations = weights
    sys.stdout = open(os.devnull, 'w', encoding="utf-8")
    for i in range(0, samples):
        game = Game(rounds, players, playerStrengths1, verbose, optimisations, best_weights)
        informedPlayerScores.append(game.getPlayers()[0].getScore())
    sys.stdout = sys.__stdout__
    print("---%s seconds---" % (time.time() - startTime))
    print("Rounds played: " + str(rounds))
    informednp = np.array(informedPlayerScores)
    print(f"informed mean: {np.mean(informednp)}")
    return np.mean(informednp)

if __name__ == "__main__":
    study = optuna.create_study(direction='maximize')
    study.optimize(objective, n_trials=200)

    print("Best trial:")
```

```
---21.155262231826782 seconds---
Rounds played: 26
informed mean: 144.405
[I 2024-04-02 12:43:34,692] Trial 19 finished with value: 144.405 and parameters: {'weight0': 0.7395684782131203, 'weight1': 0.7493006187650564, 'weight2': 0.12
72560608099192, 'weight3': 0.10721583659145403, 'weight4': 0.48837389032232903, 'weight5': 0.6329636464962389, 'weight6': 0.7635137821174083, 'weight7': 0.27581
10641020538, 'weight8': 0.37965428598119877, 'weight9': 0.6897080762375672, 'weight10': 0.1740370865687505, 'weight11': 0.2816763346567205, 'weight12': 0.683154
1002085038}. Best is trial 1 with value: 187.2.
best weights: [0.31928451299803423, 0.42161576064898926, 0.00444089479158118, 0.08722503016975182, 0.16743380139164363, 0.35031987120644087, 0.30780029783348334
, 0.16680576427097138, 0.1750740666891044, 0.06824472907286638, 0.22391711202813638, 0.38960178410236623, 0.318236374796631]
```

Figure 4.2: Screenshot of the evaluation function for self-play along with the terminal output

4.3 ISMCTS Determinisation Step

The last notable piece of code I will be covering in this report is the determinisation function. Although the implementation of the other functions in the MCTS algorithm are all notable, I cannot cover them all in this section, however, they are present in Appendix A. The determinisation function is the extension to MCTS which is used when the game has imperfect information. This function is within the “Game Tree” class, so any variables regarding the state of the game are accessible. These are conveyed previously in the design section in figure 3.14. This step is applied to the root of the game tree to expand the root using the choices produced by the hands distributed through determinisation. Without this, the hands would be empty, and new nodes could not be created.

```python
    def determinise(self, unseen):
        # Players who have played a card in the trick have 1 less card in their hand
        indent = 0
        if self.cards_played:
            indent = len(self.cards_played)
        # Select random distribution of unseen cards
        for i in range(self.players):
            # If index is equal to player index, don’t give cards
            if i == self.i:
                continue
            if indent > 0:
                random_hand = random.sample(unseen.getCards(), k=len(self.hands[self.i]) - 1)
            else:
                random_hand = random.sample(unseen.getCards(), k=len(self.hands[self.i]))
            indent -= 1
            self.hands[i] = random_hand
            # Remove selected cards from unseen
            for card in random_hand:
                unseen.removeCard(card)
```

The parameter "*unseen*" refers to the cards that the agent has not seen yet, these could be in the other opponents' hands or have not been dealt and remain in the deck. The indent variable is used to make sure that the game state has hands of different sizes if applicable, for example, if 2 players have played their cards in a 4 player game then they will have 1 less card in their hand than the other 2 players. Therefore, the indent is set to the number of cards that have already been played. The indexing in this section is challenging, the general description is that the loop goes over each player from the first one and checks based on the cards played and the position of the current player if they need an extra card. Once this is decided, the hand is produced in lines 12 and 14. This is then saved to its respective position in "$self.hands$" which is the only part of the game state that is altered in this function. This variable is a list of hands of size $self.players$.

Chapter 5

Experimentation and Evaluation

5.1 Random Agent Evaluation

In this chapter, I will outline my method of experimentation and convey the performance of the agents through different evaluation methods. This includes proving correctness as well as ranking the agents through statistical evidence. The random agent must be tested and evaluated to set the baseline for these rankings. A key function in the parent "Player" class that represents the random agent is the "getOptions" function. Every agent uses this, and it must be correct as it enforces the rule that players must match the lead suit if they have a card of the same suit.
An example of this is given in figure 5.1, where the player's hand is shown in the upper box and the cards below are the cards that lead the trick. If no cards in the hand match the lead card suit, then any card in the player's hand can be played. The arrows in this diagram show what cards can be played given the lead. I created a unit test for this function and this is provided in Appendix B along with unit tests for the related classes.
This function is the only notable part of the random agent that requires testing, the other 2 functions: "playBid" and "playOption" are trivial since they return random options from their given space of options.

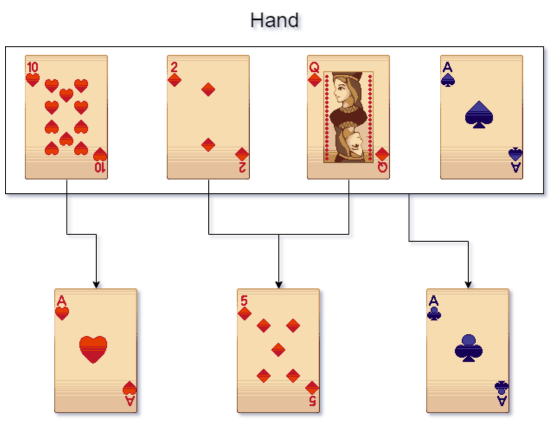


Figure 5.1: Example of how cards of the same suit as the lead card must be played

Since this is the simplest agent with no strategy, it serves as a solid baseline from which the other agents can show improvement. The experiment is fairly straightforward: games of 4 players over 13 rounds are played with decreasing hand size (from 13 to 1). To decide how many of these games were necessary to play before a good approximation was found, I ran a separate experiment to investigate the standard deviation of scores over different numbers of samples. The results of this are shown in the graph in figure 5.2. This shows all 4 players' standard deviation of score between samples 10 to 1000. It is clear that the graph plateaus, but it is difficult to see which number of samples is optimal, so I decided to average each players' standard deviation together at each number of samples, and then take the difference between each to produce the graph in figure 5.3. Using the line of best fit in this graph, I can see that the change in standard deviation remains around 0 from 300 samples and onwards. More samples are generally better, but since it takes longer to simulate more games, and the difference in the accuracy of the approximation is minimal, I believe 300 samples will suffice.

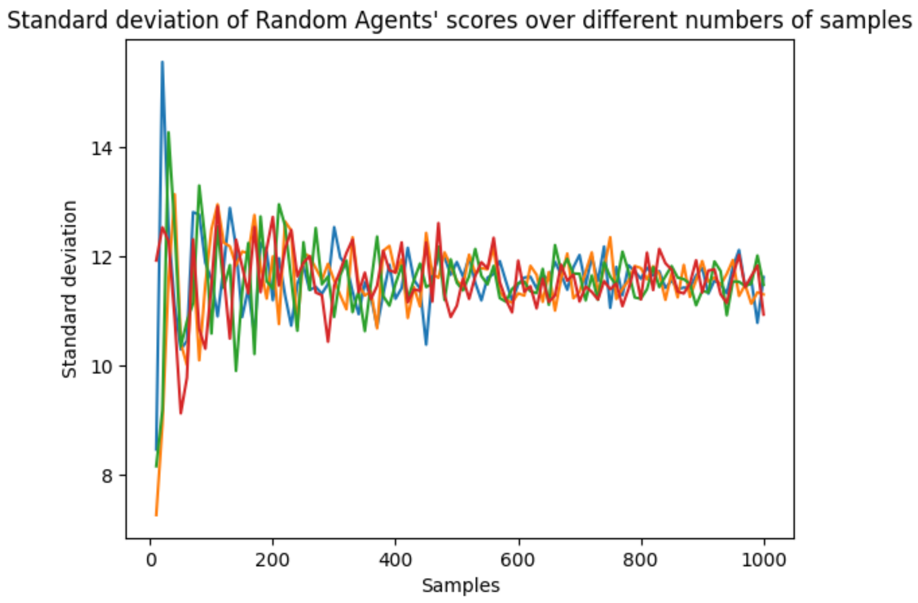


Figure 5.2: Graph showing the standard deviation of 4 different random agents' scores playing against each other over different numbers of samples

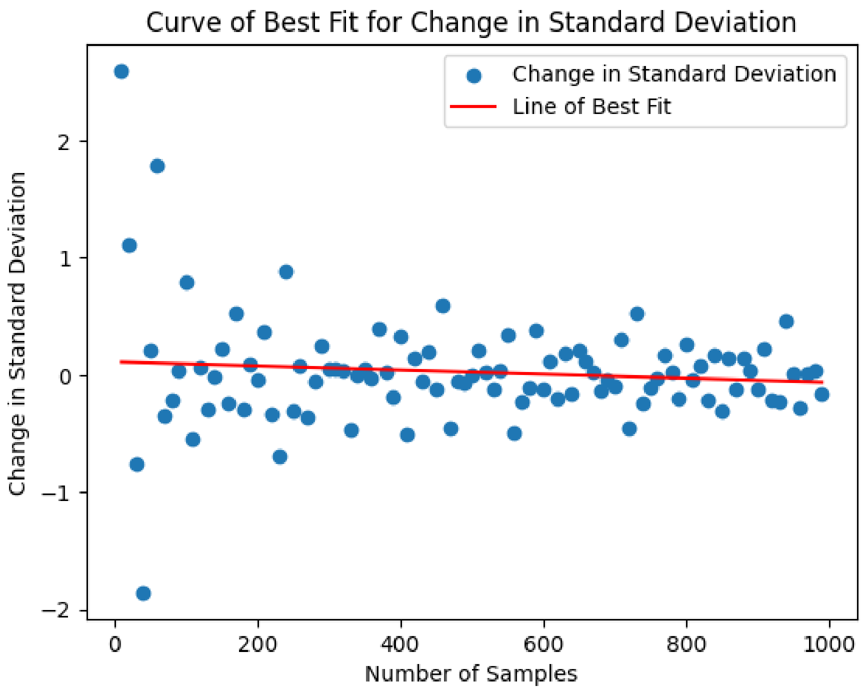


Figure 5.3: Graph showing the average change in standard deviation over all 4 agents over different numbers of samples

There are 2 factors that I believe provide metrics for good performance in "Oh Hell!". The first is the score of the player; this is an obvious choice because the player with the highest score wins the game. The next is the number of bids made per game, this directly correlates with the player's score but also gives a better understanding as to where the points are coming from. A player could receive lucky hands every round and make lots of tricks without employing much strategy or receiving their bonus from the bid. I believe a player is better when they are consistently making their bids because it shows the ability to plan and adapt to any hand. The user evaluation takes only the player's score into account. This section appears later in the chapter. Without the bids made metric, the validity of the results is hindered, however, they are still relevant. If I had more time, I would collect the number of bids made by the players in another user evaluation to further strengthen my argument.

Before evaluating against real players, I ran the previously described experiments where the agents play against each other. First is 4 random agents playing against each other. The results for the scores metric are shown in figure 5.4. This graph shows the real data points for each agent as a transparent plot as well as the bolder mean score which is shown by the solid lines. The average score between all 4 agents is rounded to 52. Many of the scores vary greatly from this mean, with the highest reaching over 90 and the lowest below 20. This is expected from a random agent which uses no decision-making.

To decide if this is a good score, we need to understand what a good score looks like. The maximum possible score an agent can achieve over one 13-round game with decreasing hand size is 221. The chance of anybody reaching this score is nearly impossible, it is extremely unlikely that a player won't lose a single trick over 13 rounds. Let us assume that 4 perfect players are playing against each other. These players can make their bid every time without fail. Under the assumption that these players play the same, we can also assume that the players' bids will roughly bid a quarter of the hand size because all 4 players are playing the same. In this ideal scenario, this gives us the approximation:

$$exceptional\_score \approx \sum_{i=1}^{rounds} \left(\frac{i}{players} + 10\right)$$

This would mean the ideal score for this experiment is approximately 153. This answer follows many assumptions but uses sound logic to provide an upper bound for the score, this can of course be exceeded with exceptional luck. A real estimate for a good score will be provided statistically from the user evaluation section later in the chapter.

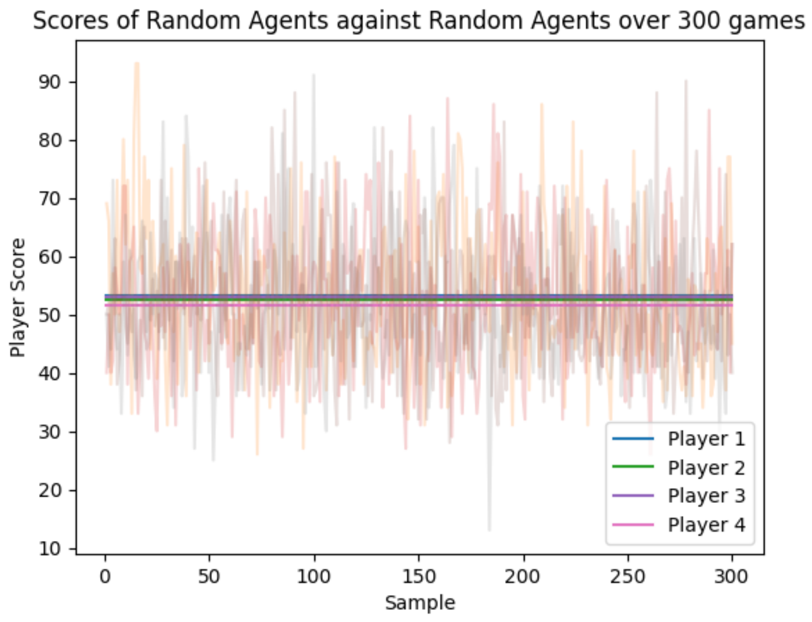


Figure 5.4: Graph showing the scores of random agents playing against other random agents over 300 simulations

The mean score being 52 is much lower than the upper bound but conclusions cannot yet be made to say if this is a "good" score in comparison to other agents and players. The graph in figure 5.5 shows the number of bids made by random agents playing against each other. Like the scores, it is clear that the random agent is much worse than the theoretical optimum of all 13 bonuses being awarded. An average of a single bid made is expected since the agent chooses a random bid each round, even if the bid is randomly accurate, the chances of the agent playing to that bid are very slim when making random choices. The graph shows samples where agents make 4 or more bids, but this is likely due to the random nature of the last few rounds that do not require much decision-making, and also have fewer bids to choose from.

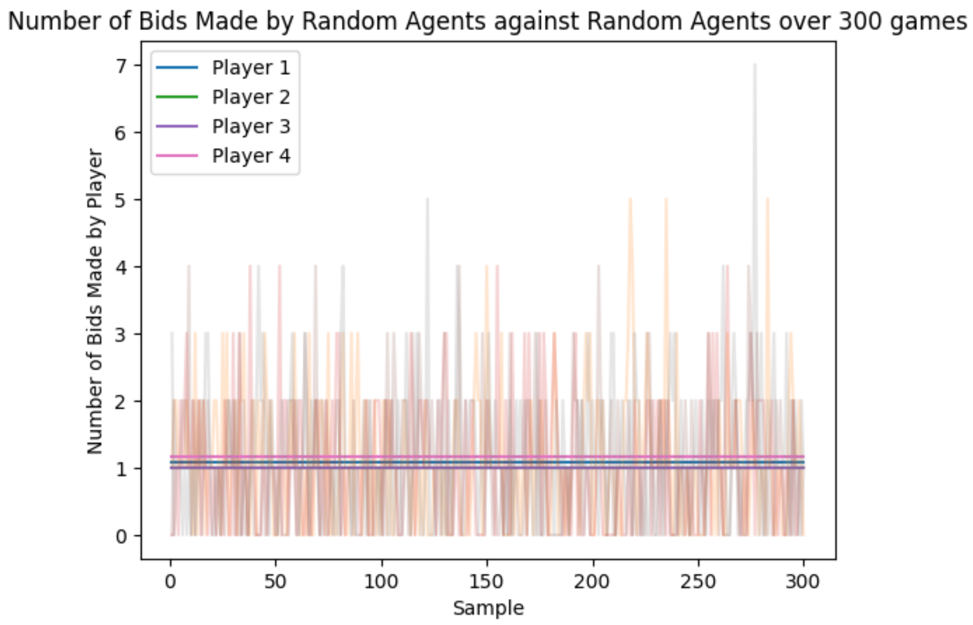


Figure 5.5: Graph showing the number of bids made by random agents playing against other random agents over 300 simulations

5.2 Informed Agent Evaluation

Testing the correctness of the informed agent is challenging because of the way the agent evaluates the differences between the values of different cards. The optimised weights make the evaluation, and consequently, the bidding algorithm unpredictable and challenging to test. However, I can prove the correctness of the algorithm that plays an option mentioned in the design chapter. This can be done by making sure the agent plays winning options when the agent is under its bid and losing options when the agent is on its bid. This is illustrated in figure 5.6 where the card highlighted in red is chosen given the card played and the state the player is in. I created unit tests for every type of scenario that can be encountered in the play option function; tests can be found in Appendix B.

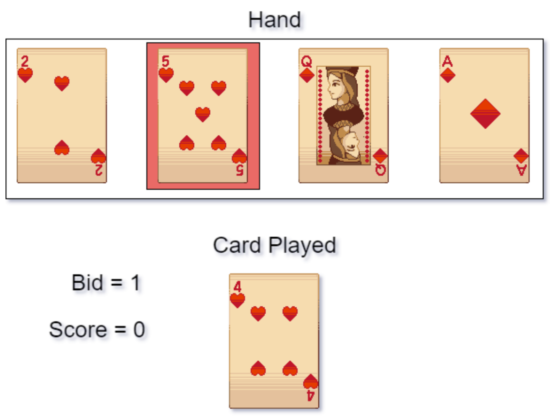


Figure 5.6: Example of how the play option algorithm wins when under bid if it can

To gauge how the informed agent compares to the random agent, I use the same experiment setup as the previous section. I believe it is necessary to not only make the informed agents play against themselves but also the random agent. This is to observe whether or not playing agents with no decision-making impacts their performance.

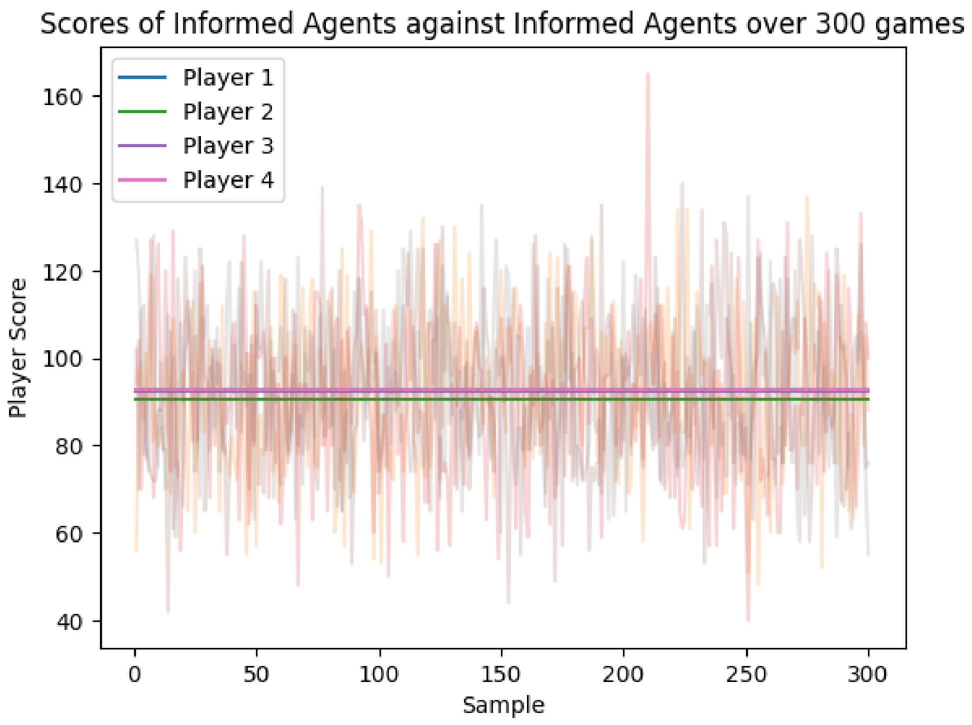


Figure 5.7: Graph showing the scores of informed agents playing against other informed agents over 300 simulations

It is immediately apparent from figure 5.7 that there is an increase in performance, the average score now comes to 92 which is much greater than the random agent's average score of 52. This 40-point increase comes from the improved bidding and execution of rounds, this is evident from the graph in figure 5.8 showing the number of bids made by informed agents playing against other informed agents. The average number of bids made has increased to 5 here, this is 4 over the random agent's 1 on average. This increase in 40 bonus points is clearly what separates the two agents. In both figures, the randomness of the game gives the agent a varied performance. For instance, in figure 5.7, there are outliers ranging from 40 to 160 points. The informed agents do not have any random elements in their decision-making. High scores with few bids made are often attributed to a combination of good cards and the play options algorithm making the agent try and win every trick once they go above their bid.

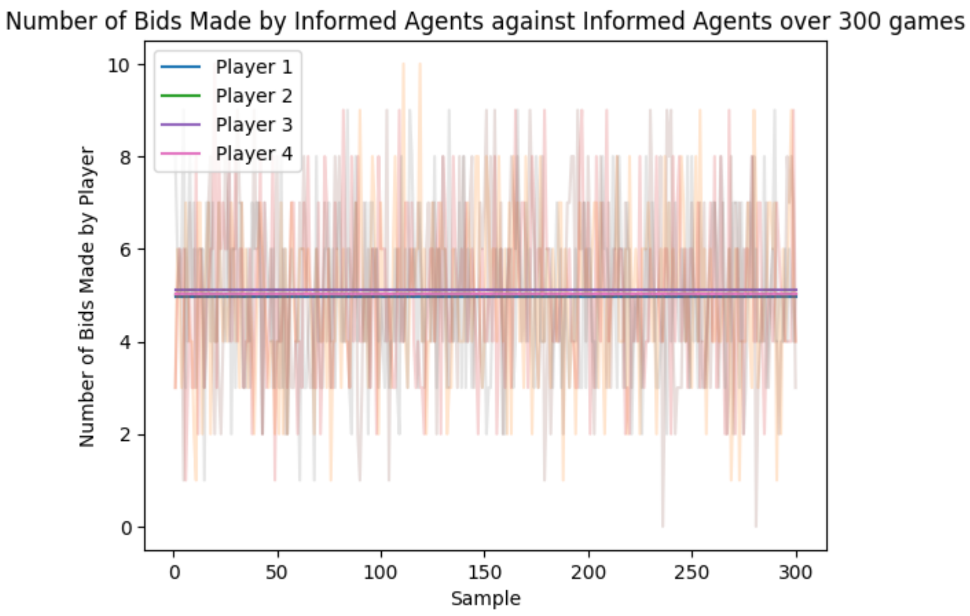


Figure 5.8: Graph showing the bids made by informed agents playing against other informed agents over 300 simulations

Next, I investigate how different agents playing against each other impacts their performance. To do this I decided to make 1 informed agent play against 3 random agents. The scores from this experiment can be seen in figure 5.9, the random agent's performed the same as they did when playing against themselves, but the single informed agent's score improved slightly to an average score of 100. This is because random agents can easily choose an option that can allow the informed agents to win more tricks. For example, if the informed agent wants to win tricks to make their bid and the informed agent played a card that can be beaten by 1 out of 4 of a random agent's cards, then a good player who also wants to win would play the card that wins. But instead of guaranteeing the win like the informed player would, the random agent has a 3 in 4 chance of losing the trick. This allows the informed player to score more points over all, but not necessarily make more bids as shown in figure 5.10, where the average bids made for both agents remain the same.

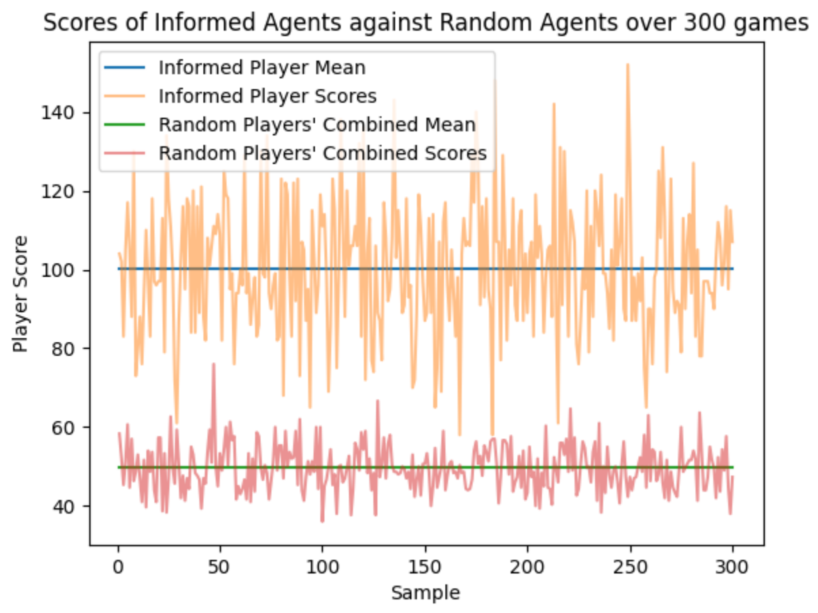


Figure 5.9: Graph showing the scores of an informed agent playing against random agents over 300 simulations

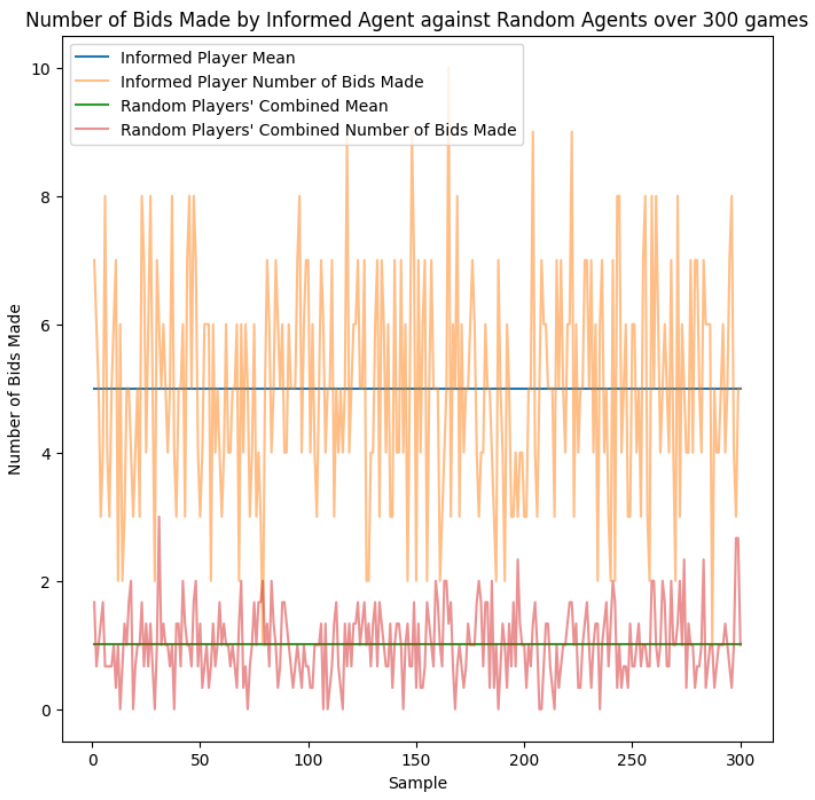


Figure 5.10: Graph showing the bids made by an informed agent playing against random agents over 300 simulations

5.3 Monte Carlo Agent Evaluation

My initial assumption about the agents' standings was that the Monte Carlo agent would excel over the informed agent, due to the informed agent's straightforward approach giving it the inability to plan ahead. This section examines and supports this hypothesis with evidence. But first, it is necessary to verify the correctness of the Monte Carlo agent, which relies on the correctness of the "GameTree" class. The results from the simulation phase in the game tree can be random due to the default policy [BPW$^+$12]. However, when using smaller hand sizes, the tree will always evaluate to the same value as in figure 3.15 which shows an example of how a game tree is built given the agent has already determinised the other agent's hands. This scenario has been made into a unit test to prove the correctness of the game tree and is provided in Appendix B.

When first planning the evaluation of the Monte Carlo agent, I came across a challenge; if I wanted to run the same experiment as the previous 2 sections, I would have to wait over 4 days for the result of the Monte Carlo agents against other Monte Carlo agents to be completed. The time taken to make a move is limited to a maximum of 5 seconds per move. Although this does not seem like a lot of time, the previous agents could make moves in a few milliseconds, providing an immense time save in comparison: taking less than 10 seconds to complete the 300 simulations in the experiments. Therefore I had to think of a way to assess the agent as well as I could given the time frame. The first thing I did was run an experiment where a single Monte Carlo agent plays against 3 random agents. I decided that only this was necessary over making the Monte Carlo agent play against itself because it takes less time and also because I've already done an experiment showing how the informed agent performs against the random agent, so the agents can easily be compared. I also made the experiment run only 20 simulations so that the experiment could run in less than 5 hours. This will impact the validity of the results, but should still give an idea of the agent's performance. The results from this experiment are shown in figure 5.11 and figure 5.12 below.

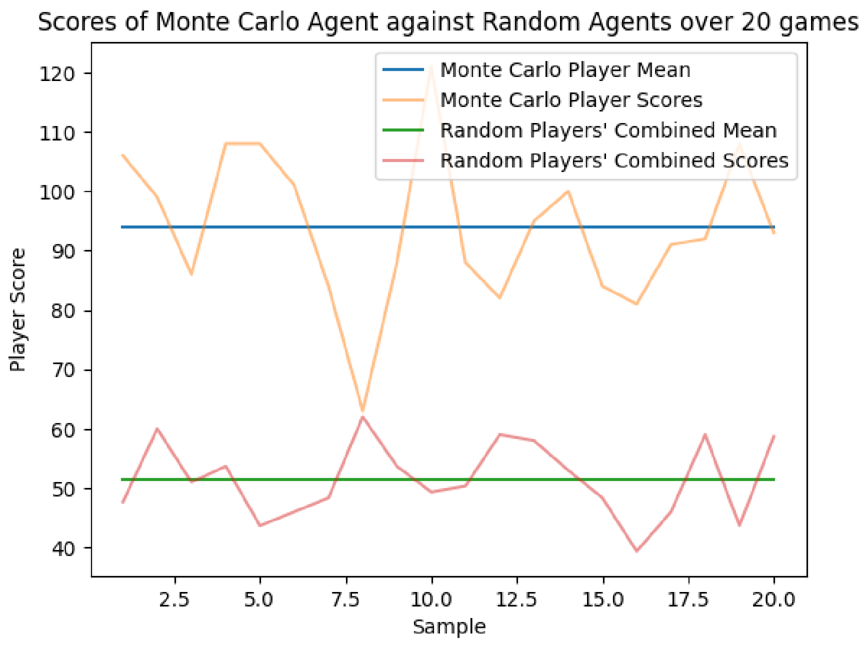


Figure 5.11: Graph showing the scores of a Monte Carlo agent playing against random agents over 20 simulations

The average score of the random agents is the same as the previous experiments as expected. Figure 5.11 shows that the Monte Carlo agents have an average score of

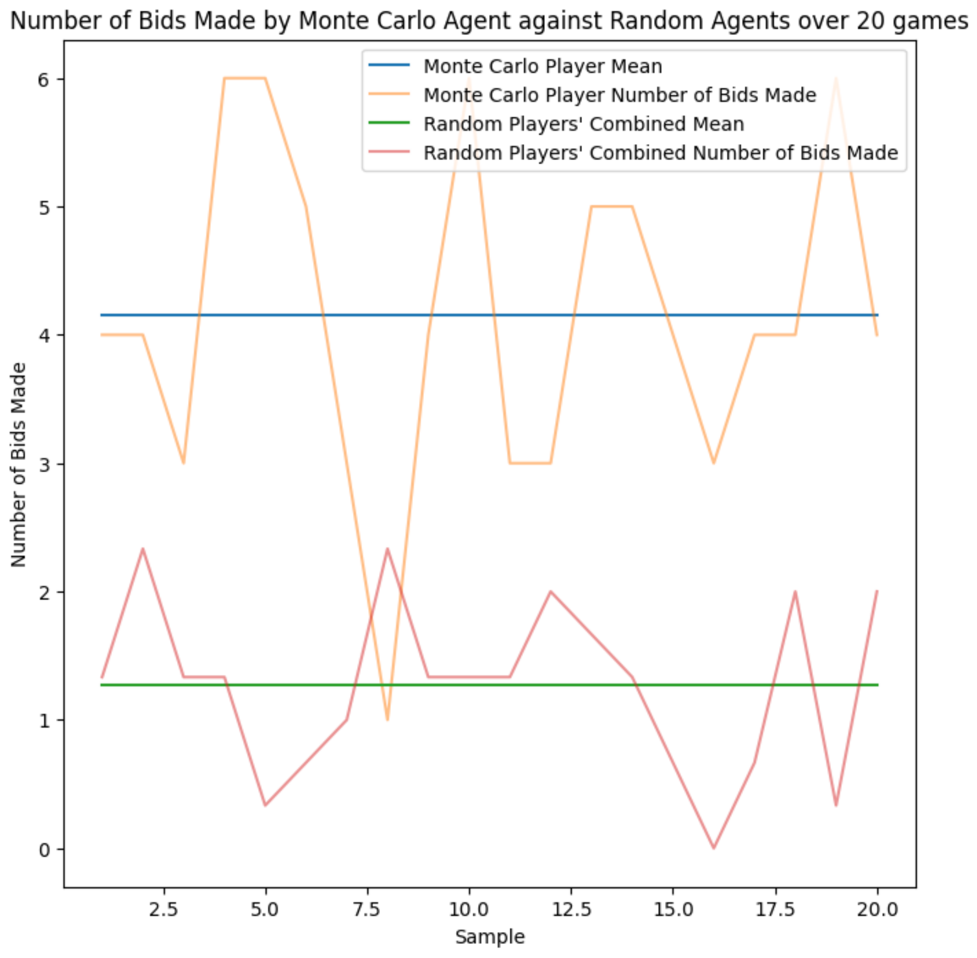


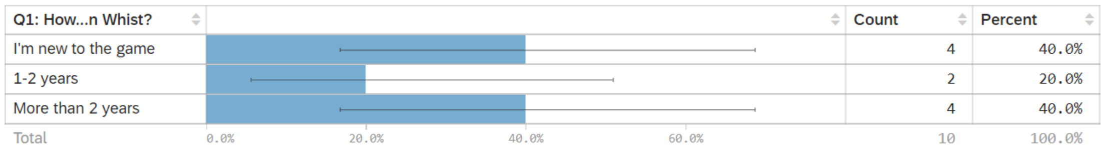
Figure 5.12: Graph showing the bids made by a Monte Carlo agent playing against random agents over 20 simulations

94, which is slightly less than the informed agent. Multiple possible factors contribute to this, firstly it is known that “UCT allows MCTS to converge to the minimax tree” [BPW+12], which means that the Monte Carlo agent is converging to a pessimistic solution. This is because the minimax algorithm chooses the maximum node of the opponent’s minimum nodes [BMP14]. Though minimax is the theoretical optimal play for the agent, it may not get as great of a score as the informed agent since the random agent will not follow the optimal play and will make poor choices in which the Monte Carlo agent could have capitalised. Another reason for the agent’s underperformance in comparison to the informed agent is in the potential need for more time to explore samples within the information set, ensuring an adequate evaluation of potential moves. The graph in figure 5.12 reflects this since the average number of bids made by the Monte Carlo agent is slightly lower than the informed agent. This is because the Monte Carlo agent will try to plan down to the last move how to make the bid, if the random agent does not play exactly what the Monte Carlo agent predicted, then the bid would be ruined. A further reason could be that the number of samples used in the Monte Carlo evaluation was not large enough to give an accurate evaluation.

To decide whether the agents have produced a "good" score, a user evaluation is required to test them against real players. With data from real players, I can verify the claim that both the informed agent and the Monte Carlo agent play well to complete my aim mentioned in the introduction. I can also uncover which of the 2 is the better agent. My approach to the collection of user data is as follows:

- Send the participant the compiled game which utilises the previously mentioned graphical user interface.
- Ask them to play 3 games with the same parameters used in the experiments in this section.
  - The first game is against 3 random agents to provide a baseline for the performance of the real players.
  - The second game is against 3 informed agents to assess the difference between random agents and informed agents.
  - The third game is against 3 Monte Carlo agents to investigate whether it is better than the informed agent against real players.
- Ask the participant to take part in a survey discussing their level of experience.

In total, 14 people took part in the evaluation, some participants played the game multiple times so the total number of samples for each difficulty was 22. This is a similar number of samples to the previous experiment and gives an overview of the performance of each agent. As mentioned, only the scores were collected from users. A total of 10 participants took part in the survey that was created on Qualtrics. From the participants who did take part in the survey, it is clear to see from figure 5.13 that they had a range of experience, some players were playing the game for the first time, whilst others had played for many years.
The inexperience of certain players can be seen in figure 5.14 which shows the scores of the participants playing against each type of agent. Each sample in this graph represents the same player's 3 scores that were sent to me upon completion. Though there



Q1: How...n Whist?		Count	Percent
I'm new to the game		4	40.0%
1-2 years		2	20.0%
More than 2 years		4	40.0%
Total	0.0% 20.0% 40.0% 60.0%	10	100.0%

Figure 5.13: Results from question 1 of the user survey asking how long participants have played the game

is no clear trend between the players' scores against different agents, there is a trend between most players' 3 entries, which follow a similar pattern where newer players tend to have lower scores in all 3 games. This tells me that results from the experiment vary between individuals.

The average score of real players against the easy agent is 100. The setup for this experiment is the same as the experiments that evaluate the informed agent and Monte Carlo agent against random agents in figure 5.9 and figure 5.11 respectively. The average score of the informed agent is 100, and the Monte Carlo agent is 94. So the agents perform similarly to real players against the random agent. This does not confirm that the agents are as good as real people since the random agents have no decision-making, the agents could perform more poorly than real players against opponents with good decision-making. The average score of the informed agent versus itself in figure 5.7 can be compared against the average score of participants against the informed agent. Both average scores are the same again at 92 points. However, the average score of the informed agents who played against real players is 73 shown in figure 5.15. This decrease in average score could be because of participants realising the obvious strategy that the informed agent utilises. The informed agent tends to play its highest-valued cards first because, when it is underbid, it tries to maximise its chances of winning the trick. This can be exploited by players who know of this habit, by playing your low cards first, a player can ensure they win tricks later on since all the opponent's best cards will have already been played. To confirm this I would have to add this as a question to an additional survey. Figure 5.15 shows the best argument for why the Monte Carlo agent is the best performing agent out of the 3 agents. This is because the Monte Carlo agents perform better against real players on average than informed agents. The Monte Carlo agent's mean score is 82 whilst the informed agent's mean score is 73. I believe the informed agent's recognizable habit stood out, unlike the Monte Carlo agent which has no characteristic strategies because of its random nature.

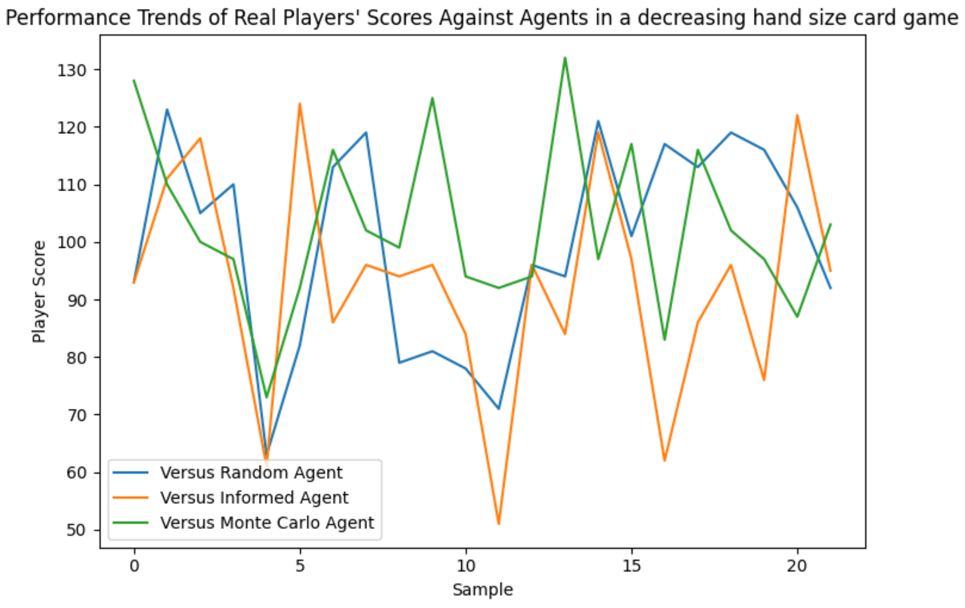


Figure 5.14: Graph showing the scores of participants against each type of agent over 22 samples

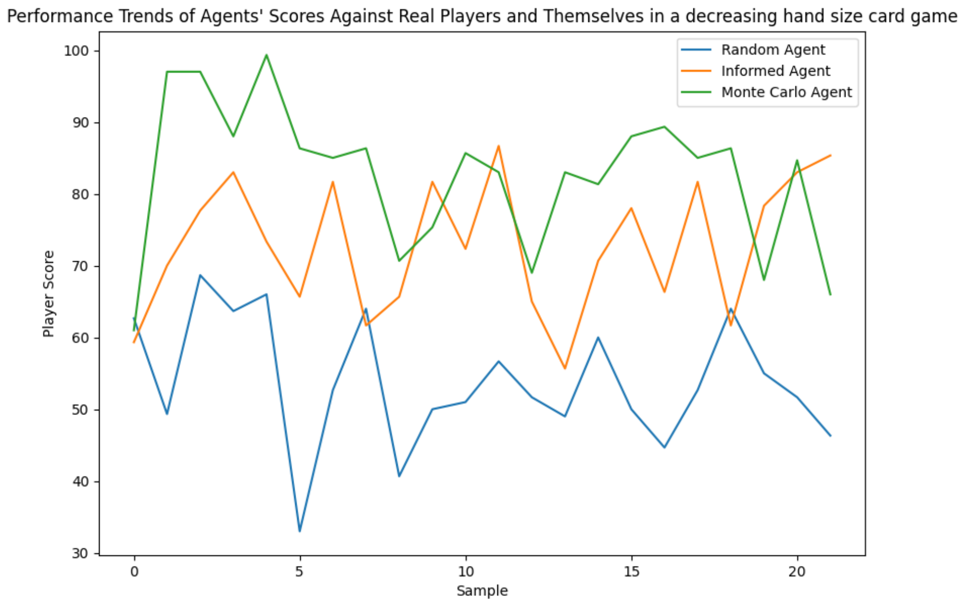


Figure 5.15: Graph showing the average scores of agents against participants over 22 samples

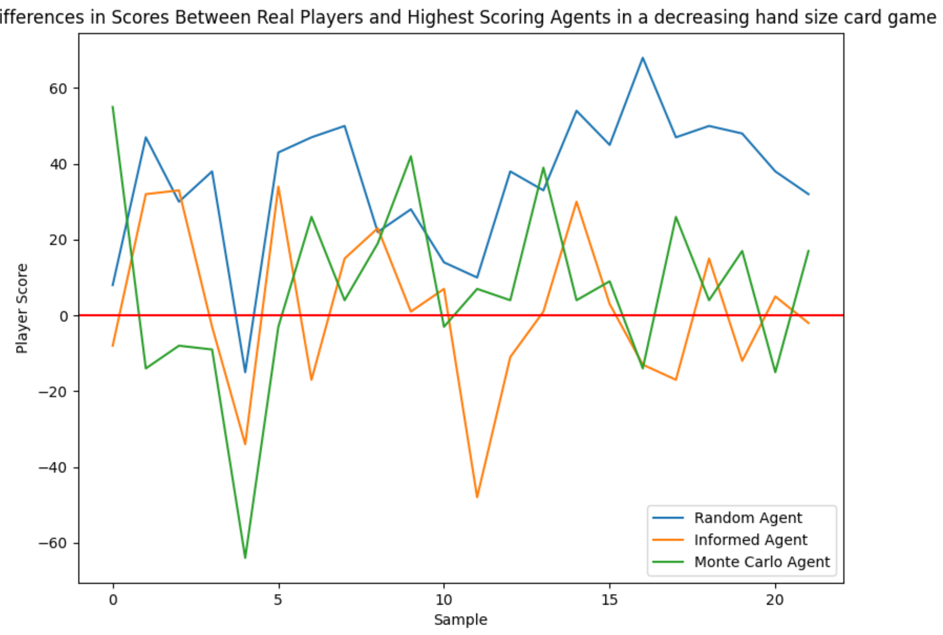


Figure 5.16: Graph showing the difference between scores of the participant and the highest scoring agent in each game

Figure 5.16 was made to investigate which agents won the most games and by how much. Points below the red line show games where the agents have won, and points above show games where the player has won. The informed agent won the most games out of any other agent, with the Monte Carlo agent in a close second. This could be because players were forced to spend longer to think about their moves while waiting for the Monte Carlo agent to make a move, or because many participants did not discover the characteristic strategy of the informed agent before the end of the game.

I believe the Monte Carlo agent performed the best overall. This is because of its ability to plan ahead and consider different possible outcomes before making the best move based on its judgment. The informed agent takes much less time but is impeded by its play style and lack of forward planning. This partially proves my initial hypothesis stated at the beginning of the section; however, further evidence is necessary to minimise the randomness of the results and fully substantiate my claim.
To decide if the agent is "expert level" as mentioned in the introduction, it would be logical to compare the average scores of the agents against the scores from real players. This can be done using a histogram like the one in figure 5.17. After counting all the

scores that the agents beat, I can conclude that:

- The random agent is in the top 98% of players
- The informed agent is in the top 92% of players
- The Monte Carlo is in the top 83% of players

To be an expert player, one would expect the agent to be above the top 50% of players at least, meaning that the results indicate the agents are below average level. Players with years of experience sent in multiple samples to the user evaluation, possibly skewing the portrayed average skill level. With further optimisations of the Monte Carlo agent, and by combining the best aspects of both agents, I believe an expert-level agent could be achieved with more time.

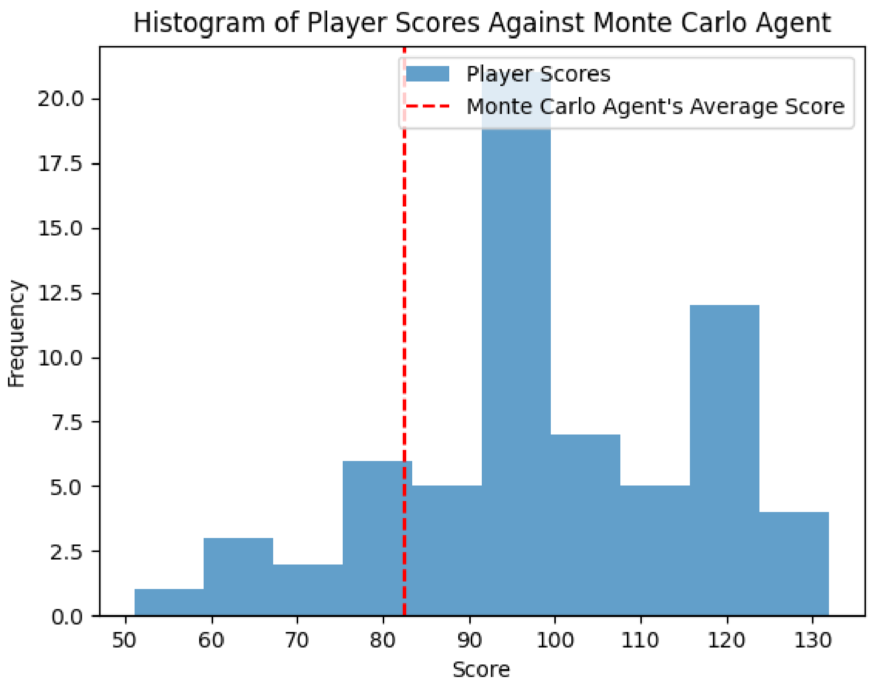


Figure 5.17: Histogram of the scores of participants

Chapter 6

Conclusions

6.1 Summary of Work and Discussion of Results

Throughout this report I have given context about the domain of AI in card playing, provided an in-depth description of the process of designing the game and the agents, shown notable snippets of code that posed implementation challenges, and experimented with the agents to investigate their capabilities. The output of this project has been the game of “Oh Hell!” as a piece of software with 3 different difficulty modes. Whilst developing these difficulty modes, I have met my aim of exploring different techniques in game theory and discussed the different methods that can be taken to create an AI for “Oh Hell!”. The method outlined can be extended to many card games with a similar structure like bid whist, bridge, hearts, etc.

Ranking the agents, the random agent is automatically the worst. It has no decision-making at all and is mostly included as a benchmark the other agents can be compared to. The informed agent is next, this knowledge-based approach allowed for heuristics from my experience to dictate the value of cards based on the state of the game. However, these heuristics are not the only source used to decide between cards, the algorithms for playing a move and bidding are naive and follow a very simple set of rules that do not allow the agent to plan its future moves. On top of this, the informed agent tends to play its highest-valued cards first, this means that any observant players can use its strategy against it. If the informed agent had done self-play with agents that worked against its current strategy, it may have obtained an improved strategy devoid of this characteristic. Though the informed agent beat the Monte Carlo agent when evaluated against the random agent, the Monte Carlo agent showed much better results against real players from the user evaluation. The Monte Carlo agent was slow and difficult to evaluate effectively, but unlike the previous agents, it planned future moves and evaluated different actions without any additional information.

The agents produced play to a novice level, with the Monte Carlo agent being the best beating 17% of the players in my study. My original aim was to create an "expert level" agent that could beat at least 50% of players. Many results from the user evaluation were produced by players who have a considerable amount of experience with the game. This may have portrayed the agents as underperforming compared to their potential when assessed against a diverse group of participants with varying skill levels. Nonetheless, I believe the agents can still be further optimised to attain an expert level of play as I originally aimed to.

6.2 Further Work

From all that I have learned from working on this project, I believe I would be able to produce a superior agent from those I have already made that employs a combination of the different techniques mentioned in this paper. By utilising the heuristics and self-play techniques from the informed agent, I can improve the speed of the ISMCTS algorithm and further improve other hyper-parameters such as the exploration weight of the UCT algorithm. This is similar to the idea of Whitehouse et al. [WCPR13] where ISMCTS is implemented along with knowledge-based methods to make an agent that could play Spades. This agent was released on a mobile platform and played thousands of games against real players, which is something I would consider if I had more time. By doing this, I would have a great deal of data to aid in the optimisation of the Monte Carlo agent and be able to make a neural network to aid the agent, resembling how AlphaGo used a deep neural network in tandem with a modified version of MCTS to develop their exceptional agent [SHM$^+$16].

If I were to code the game and agents again, the structure of my implementation would be much cleaner and more concise. This would allow for the "special rounds" that were mentioned in the introduction to be integrated into the game and the agents. Extra care would also have been taken when designing the user interface for the user evaluation, many participants made helpful comments about how it could be improved in my survey.

Bibliography

[ASY$^+$19a] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[ASY$^+$19b] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna documentation, 2019. Accessed on 23/02/2024.

[BMP14] Lucian Buşoniu, Rémi Munos, and Elő̋d Páll. An analysis of optimistic, best-first search for minimax sequential decision making. In *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 1–8. IEEE, 2014.

[BPW$^+$12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[CWP15] Peter I Cowling, Daniel Whitehouse, and Edward J Powley. Emergent bluffing and inference with monte carlo tree search. In *2015 IEEE conference on computational intelligence and games (CIG)*, pages 114–121. IEEE, 2015.

[DR11] Laurent Doyen and Jean-François Raskin. Games with imperfect information: theory and algorithms. *Lectures in Game Theory for Computer Scientists*, 10, 2011.

[HWCH18] Chu-Hsuan Hsueh, I-Chen Wu, Jr-Chang Chen, and Tsan-sheng Hsu. Alphazero for a non-deterministic game. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 116–121. IEEE, 2018.

[Lun99] Fredrik Lundh. An introduction to tkinter. *URL: www. pythonware. com/library/tkinter/introduction/index. htm*, 1999.

[NAP^{+}19] Joel Niklaus, Michele Alberti, Vinaychandran Pondenkandath, Rolf Ingold, and Marcus Liwicki. Survey of artificial intelligence for card games and its application to the swiss game jass. In *2019 6th Swiss Conference on Data Science (SDS)*, pages 25–30. IEEE, 2019.

[Ope08] MIT OpenCourseWare. Cms. 608/cms. 864 game design. 2008.

[Par08] D. Parlett. *The Penguin Book of Card Games: Everything You Need to Know to Play Over 250 Games*. Penguin Publishing Group, 2008.

[SHM^{+}16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SW07] Nathan R Sturtevant and Adam M White. Feature construction for reinforcement learning in hearts. In *Computers and Games: 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers 5*, pages 122–134. Springer, 2007.

[T^{+}95] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[txt15] txturs. Grass pixel art. Image, 2015. https://opengameart.org/content/grass-pixel-art.

[WCPR13] Daniel Whitehouse, Peter Cowling, Edward Powley, and Jeff Rollason. Integrating monte carlo tree search with knowledge-based methods to create engaging play in a commercial mobile game. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 9, pages 100–105, 2013.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger*, 1(1):7–87, 1990.

[Wol21] Caz Wolf. Pixel fantasy playing cards. Image, 2021. https://cazwolf.itch.io/pixel-fantasy-cards.

Appendix A

MCTS Implementation

```python
import random
import math
import time
import numpy as np
import copy
from collections import defaultdict

class GameTree:

    def __init__(self, parent = None, hands = [], card_choice
        = None, cards_played = [], scores = [], bids = None,
        players = None, trump = None, current_player = 0,
        max_depth = None, depth = 0) -> None:
        self.parent = parent
        self.children = []
        self.hands = hands # cards player has
        self.cards_played = cards_played # cards that have
            been played in the trick
        self.scores = scores # current scores of players (in
            order)
        self.bids = bids # current bids of players (in order)
        self.players = players # number of players
        self.trump = trump # trump suit for the round
        self.i = current_player # player whose turn it
            currently is
        self.max_depth = max_depth
        self.depth = depth
        self.visits = 0 # times node was visited
        self.wins = 0 # score of node
        self.choices = []
        self.terminate = False
        self.card_choice = card_choice
        if len(self.cards_played) == self.players or not self.cards_played:
            self.choices = self.hands[self.i]
        else:
            self.choices = self.get_choices(self.hands[self.i], self.cards_played[0])
        self.unprocessed_choices = copy.deepcopy(self.choices)
        if self.depth >= self.max_depth or not self.choices: # if depth is exceeded or player has no moves (round finished) then return
            return

    def get_choices(self, hand, leadCard):
        choices = []
        for card in hand: #add cards of the same suit as the lead to the options
            if card.getSuit() == leadCard.getSuit():
                choices.append(card)
        if not choices: #if no cards with the same suit as the lead, all cards are options
            choices = hand
        return choices

    def determinize(self, cardsInDeck):
        unseen = copy.deepcopy(cardsInDeck)
        # already played a card in trick so must deduct cards from initial players hands
        indent = 0
        if self.cards_played:
            indent = len(self.cards_played)
        # select random distribution of unseen cards for other players to have
        for i in range(self.players):
            if i == self.i:
                continue
            if indent > 0:
                random_hand = random.sample(unseen.getCards(), k=len(self.hands[self.i]) - 1)
            else:
                random_hand = random.sample(unseen.getCards(), k=len(self.hands[self.i]))
            indent -= 1
            self.hands[i] = random_hand
            # remove selected cards from unseen
            for card in random_hand:
                unseen.removeCard(card)

    def select_child(self, exploration_weight=2, threshold = 1000):
        """Select a child node using UCT (Upper Confidence Bound for Trees) formula"""
        # if no children, select itself
        if not self.children:
            return self
        log_total_visits = 0
        for child in self.children:
            log_total_visits += child.visits
        if self.visits > threshold:
                self.terminate = True
                return self
        log_total_visits = math.log(log_total_visits)
        # if all choices do not have a node, select itself
        if len(self.children) < len(self.choices):
            return self

        def uct_value(child):
```

```python
            exploitation_term = child.wins / child.visits
            exploration_term = exploration_weight * math.sqrt((2*log_total_visits) / child.visits)
            ##print(exploitation_term + exploration_term)
            return exploitation_term + exploration_term

        # Choose the child with the maximum UCT value
        child_max = max(self.children, key=uct_value)
        if self.unprocessed_choices:
            return self
        if child_max.children:
            child_max = child_max.select_child()
        return child_max

    def expand(self):
        """Expand the node by adding a new child"""
        # create the child nodes of the current state based on the determinized game
        # update the cards played
        if not self.choices or self.depth >= self.max_depth:
            return self

        if self.unprocessed_choices:
            new_choice = self.unprocessed_choices.pop(0)  # process in order so they can be easily indexed by the choosing agent
        else:
            return self
        new_hands = copy.deepcopy(self.hands)
        new_scores = copy.deepcopy(self.scores)
        next_player = copy.deepcopy(self.i)
        new_hand = []
        for card in self.hands[self.i]:
            if not card.equalTo(new_choice):
                new_hand.append(card)
        new_hands[self.i] = new_hand

        next_player +=1
        if next_player == self.players:
            next_player = 0
        if len(self.cards_played) == self.players:
            cards_played = []
        else:
            cards_played = copy.deepcopy(self.cards_played)
        cards_played.append(new_choice)
        if len(cards_played) == self.players:
            for i, card1 in enumerate(cards_played):
                # Assume player i has won until proven
                   otherwise
                player_wins = True
                for card2 in cards_played:
                    if card1.equalTo(card2):
                        continue
                    if not card1.beats(card2,
                       cards_played[0].getSuit(), self.trump):
                        # Card1 does not beat card2, set
                           player_wins to False and break the
                           inner loop
                        player_wins = False
                        break
                if player_wins:
                    # All comparisons passed, player i wins
                    next_player = (next_player + i) % 4
                    new_scores[next_player] += 1
                    break

        child = GameTree(self, new_hands, new_choice,
           cards_played, new_scores, self.bids, self.players,
           self.trump, next_player, self.max_depth,
           self.depth + 1)
        self.children.append(child)
        return child

    def simulate(self):
        """Simulate a random playout from a expanded node"""
        # copy variables to send to random playout
        hands = copy.deepcopy(self.hands)
        scores = copy.deepcopy(self.scores)
        player = copy.deepcopy(self.i)
        cards_played = copy.deepcopy(self.cards_played)
        simulation_val = self.simulate_random_playout(hands, scores, player, cards_played)
        return simulation_val

    def simulate_random_playout(self, hands, scores, player, cards_played):
        """Simulation of random moves from the current cards"""
        while True: # until there are no more random moves to make, keep making random moves
            if len(cards_played) == self.players or not cards_played: # get player choices
                choices = copy.deepcopy(hands[player])
            else:
                choices = self.get_choices(hands[player], cards_played[0])
            if not choices: # if no choices left then round is over
                #print(scores)
                return self.evaluate(scores) # return the evaluation of the final state
            if type(choices) is not list:
                choices = [choices]
            if self.bids:
                if self.bids[player] == scores[player] and len(cards_played) != self.players and len(choices) != 1: # Optimisation: if player can avoid winning a trick to stay on bid then do so
                    for c in choices:
                        wins = True
                        for card in cards_played:
```

```
                        if not c.beats(card, cards_played[0].getSuit(), self.trump):
                            wins = False
                            break
                    if wins == True: # remove choices that could cause a win when bid is met
                        choices.remove(c)
            if len(choices) == 0:
                if len(cards_played) == self.players or not cards_played: # get player choices
                    choices = hands[player]
                else:
                    choices = self.get_choices(hands[player], cards_played[0])
                if type(choices) is not list:
                    choices = [choices]
        choice = random.choice(choices) # pick random choice
        # #print(f"player {player + 1}: {choice}")
        hands[player] = [card for card in hands[player] if not card.equalTo(choice)] # remove choice from players hand
        if len(cards_played) == self.players:
            cards_played = []
        cards_played.append(choice) # add choice to cards played
        player += 1
        if player == self.players:
            player = 0
        if len(cards_played) == self.players:
            for i, card1 in enumerate(cards_played):
                # Assume player i has won until proven otherwise
                player_wins = True
                for card2 in cards_played:
                    if card1.equalTo(card2):
                        continue
                    if not card1.beats(card2, cards_played[0].getSuit(), self.trump):
                        # Card1 does not beat card2, set player_wins to False and break the inner loop
                        player_wins = False
                        break
                if player_wins:
                    # All comparisons passed, player i wins
                    player = (player + i) % self.players
                    scores[player] += 1
                    break

def evaluate(self, scores):
    if self.bids: # if not in the bidding phase
        for i in range(self.players): # if player i made their bid award bonus points
            if self.bids[i] == scores[i]:
                scores[i] += 10
    return scores


def backpropagate(self, result):
    """Update the wins and visits for this node and its ancestors"""
    current_node = self
    while current_node.parent:
        current_node.visits += 1
        current_node.wins += result[current_node.parent.i] # wins of the node should reflect parent's choice
        current_node = current_node.parent
    return current_node

```

```python
    def __str__(self, depth):
        """Returns the tree as a tree structure in depth-first order"""
        def hand_as_string(hand):
            hand_str = ""
            for card in hand:
                hand_str += card.__str__()
            return hand_str
        output = ""
        # Add spaces based on depth
        indentation = " " * (self.depth * 4)

        # Add symbols to represent the tree structure
        if self.parent:
            if self.parent.children[-1] == self:
                line_symbol = "| "
            else:
                line_symbol = "|-- "
            output += f"{indentation}{line_symbol}"
        #output += f"depth={self.depth} | wins={self.wins} | visits={self.visits} | children={len(self.children)} | player={self.i + 1} | scores={self.scores}\n"
        # output += f"{indentation}|-hand = {hand_as_string(self.hands[self.i])}\n"
        if self.parent:
            if self.visits != 0:
                output += f"player {self.parent.i + 1} plays card {self.card_choice} = {self.wins}, {self.visits}\n"
            else:
                output += f"player {self.parent.i + 1} plays card {self.card_choice} = 0\n"
        else:
            output += f"--root--\n"
        # output += f"{indentation}|-choices = {hand_as_string(self.choices)}\n"

        # for hand in self.other_players_hands:
        #     output += f"{indentation}|-next_hand =
            {hand_as_string(hand)}\n"


        for child in self.children:
            if child.depth > depth:
                break
            output += child.__str__(depth)

        return output
```

Appendix B

Unit tests

```
import unittest
from Player import Player
from Card import Card
from Deck import Deck
from GameTree import GameTree

class TestGametree(unittest.TestCase):

    def testGametree1(self):
        players = 4
        bids = [0,1,1,2]
        current_player = 0
        iterations = 8
        hands = [[Card(9,0),Card(2,1)],
                    [Card(14,0),Card(6,2)],
                    [Card(13,0),Card(14,1)],
                    [Card(12,0),Card(14,2)]]
        wins = [0 for _ in hands[current_player]]
        cardsPlayed = []
        scores = [0,0,0,0]
        trump = 0
        game_tree = GameTree(parent = None, hands = hands,
           cards_played=cardsPlayed, players = players, bids
           = bids, scores = scores, trump=trump,
           current_player=current_player, max_depth=12)
        for _ in range(iterations):
            selection = game_tree.select_child()
            if selection.terminate:
                break
            expansion = selection.expand()
            simulated_value = expansion.simulate()
            expansion.backpropagate(simulated_value)
        for x, child in enumerate(game_tree.children):
            wins[x] += child.wins/child.visits
        # Both cards cannot win when leading and therefore
           both make bid
        self.assertEqual(wins[0] == 10 and wins[1] == 10,
           True)

    def testGametree2(self):
        players = 4
        bids = [0,1,1,2]
        current_player = 1
        iterations = 10
        hands = [[Card(9,0),Card(2,1)],
                    [Card(14,0),Card(6,2)],
                    [Card(13,0),Card(14,1)],
                    [Card(12,0),Card(14,2)]]
        wins = [0 for _ in hands[current_player]]
        cardsPlayed = []
        scores = [0,0,0,0]
        trump = 0
        game_tree = GameTree(parent = None, hands = hands,
           cards_played=cardsPlayed, players = players, bids
           = bids, scores = scores, trump=trump,
           current_player=current_player, max_depth=12)
        for _ in range(iterations):
            selection = game_tree.select_child()
            if selection.terminate:
                break
            expansion = selection.expand()
            simulated_value = expansion.simulate()
            expansion.backpropagate(simulated_value)
        for x, child in enumerate(game_tree.children):
            wins[x] += child.wins/child.visits
            print(wins[x])
        print(game_tree.__str__(5))
        # Can randomly visit 6 spades and get a loss after 3
           visits before not visiting again
        # Or playout is a loss on first visit and not visited
           again
        self.assertEqual(wins[0] == 11 and wins[1] == 0 or
           wins[1] == (22/3), True)


    def testGametree3(self):
        players = 4
        bids = [0,1,1,2]
        current_player = 2
        iterations = 8
        hands = [[Card(9,0),Card(2,1)],
                    [Card(14,0),Card(6,2)],
                    [Card(13,0),Card(14,1)],
                    [Card(12,0),Card(14,2)]]
        wins = [0 for _ in hands[current_player]]
        cardsPlayed = []
        scores = [0,0,0,0]
        trump = 0
        game_tree = GameTree(parent = None, hands = hands,
           cards_played=cardsPlayed, players = players, bids
           = bids, scores = scores, trump=trump,
           current_player=current_player, max_depth=12)
        for _ in range(iterations):
            selection = game_tree.select_child()
            if selection.terminate:
                break
            expansion = selection.expand()
            simulated_value = expansion.simulate()
            expansion.backpropagate(simulated_value)
        for x, child in enumerate(game_tree.children):
            wins[x] += child.wins/child.visits
        # King of hearts always loses, Ace of diamonds
           depends on players 2 and 4 not trumping
        self.assertEqual(wins[0] == 0, True)


    def testGametree4(self):
        players = 4
        bids = [0,1,1,2]
        current_player = 3
        iterations = 10
        hands = [[Card(9,0),Card(2,1)],
                    [Card(14,0),Card(6,2)],
                    [Card(13,0),Card(14,1)],
                    [Card(12,0),Card(14,2)]]
        wins = [0 for _ in hands[current_player]]
        cardsPlayed = []
        scores = [0,0,0,0]
        trump = 0
        game_tree = GameTree(parent = None, hands = hands,
           cards_played=cardsPlayed, players = players, bids
           = bids, scores = scores, trump=trump,
           current_player=current_player, max_depth=12)
        for _ in range(iterations):
            selection = game_tree.select_child()
            if selection.terminate:
                break
            expansion = selection.expand()
            simulated_value = expansion.simulate()
            expansion.backpropagate(simulated_value)
        for x, child in enumerate(game_tree.children):
            wins[x] += child.wins/child.visits
        # Always wins 1 from Queen of hearts since player 1
           wins and then leads a spade
        # Ace of spade lead can win but more likely to get
           trumped
        self.assertEqual(wins[0] == 1, True)

if __name__ == '__main__':
    unittest.main()
```

```
import unittest
from InformedPlayer import InformedPlayer
from Card import Card
from Deck import Deck
from Hand import Hand

class TestInformedPlayer(unittest.TestCase):

    def testWinWhenUnderBid(self):
        hand = Hand([Card(2, 0), Card(5, 0), Card(12, 1),
           Card(14, 1)])
        player = InformedPlayer("player", [0.285903516922823,
           0.1343298885883228, 0.2513171910666505,
           0.2654805206104466, 0.062968882811757O3,
           0.19781578320406684, 0.40322548219933846,
           0.1057952046865951, 0.29316352990999955,
           0.24951563665051385, 0.25987660163497556,
           0.23694603672039563, 0.25366172499411493])
        player.makeHand(hand)
        options = player.getOptions(0)
        player.makeBid(1)
        player_choice = player.playOption(options,
           [Card(4,0)], 0, 4, None, None, None)
        self.assertEqual(Card(5,0).equalTo(player_choice),
           True)

    def testLoseWhenOnBid(self):
        hand = Hand([Card(2, 0), Card(5, 0), Card(12, 1),
           Card(14, 1)])
        player = InformedPlayer("player", [0.285903516922823,
           0.1343298885883228, 0.2513171910666505,
           0.2654805206104466, 0.06296888281175703,
           0.19781578320406684, 0.40322548219933846,
           0.1057952046865951, 0.29316352990999955,
           0.24951563665051385, 0.25987660163497556,
           0.23694603672039563, 0.25366172499411493])
```

```python
        player.makeHand(hand)
        options = player.getOptions(1)
        player.addRoundScore(1)
        player.makeBid(1)
        player_choice = player.playOption(options, [Card(13,1)], 0, 4, None, None, None)
        self.assertEqual(Card(12,1).equalTo(player_choice), True)

    def testForcedWinWhenOnBid(self):
        hand = Hand([Card(2, 0), Card(5, 0), Card(12, 1), Card(14, 1)])
        player = InformedPlayer("player", [0.285903516922823, 0.1343298885883228, 0.2513171910666505, 0.2654805206104466, 0.06296888281175703, 0.19781578320406684, 0.40322548219933846, 0.1057952046865951, 0.29316352990999955, 0.24951563665051385, 0.25987660163497556, 0.23694603672039563, 0.25366172499411493])
        player.makeHand(hand)
        options = player.getOptions(1)
        player.addRoundScore(1)
        player.makeBid(1)
        player_choice = player.playOption(options, [Card(11,1)], 0, 4, None, None, None)
        self.assertEqual(Card(14,1).equalTo(player_choice) or Card(12,1).equalTo(player_choice), True)

    def testForcedLoseWhenUnderBid(self):
        hand = Hand([Card(2, 0), Card(5, 0), Card(12, 1), Card(14, 1)])
        player = InformedPlayer("player", [0.285903516922823, 0.1343298885883228, 0.2513171910666505, 0.2654805206104466, 0.06296888281175703, 0.19781578320406684, 0.40322548219933846, 0.1057952046865951, 0.29316352990999955, 0.24951563665051385, 0.25987660163497556, 0.23694603672039563, 0.25366172499411493])
        player.makeHand(hand)
        options = player.getOptions(0)
        player.makeBid(1)
        player_choice = player.playOption(options,
           [Card(14,0)], 0, 4, None, None, None)
        self.assertEqual(Card(2,0).equalTo(player_choice),
           True)




if __name__ == '__main__':
    unittest.main()
```

```python
import unittest
from Player import Player
from Card import Card
from Deck import Deck
from Hand import Hand

class TestPlayer(unittest.TestCase):

    def testgetOptions(self):
        hand = Hand([Card(10, 0), Card(2, 1), Card(12, 1),
           Card(14, 2)])
        player = Player("player")
        player.makeHand(hand)
        options = player.getOptions(0)
        self.assertEqual(Card(10,0).equalTo(options[0]), True)
        options = player.getOptions(1)
        self.assertEqual(Card(2,1).equalTo(options[0]), True)
        self.assertEqual(Card(12,1).equalTo(options[1]), True)
        options = player.getOptions(3)
        self.assertEqual(Card(10,0).equalTo(options[0]), True)
        self.assertEqual(Card(2,1).equalTo(options[1]), True)
        self.assertEqual(Card(12,1).equalTo(options[2]), True)
        self.assertEqual(Card(14,2).equalTo(options[3]), True)

```

```python


if __name__ == '__main__':
    unittest.main()
```

```python
import unittest
from Hand import Hand
from Card import Card

class TestHand(unittest.TestCase):

    def testRemove(self):
        card1 = Card(3, 0)
        card2 = Card(5, 0)
        card3 = Card(9, 1)
        cardList = []
        cardList.append(card1)
        cardList.append(card2)
        cardList.append(card3)
        hand1 = Hand(cardList)
        cardList.remove(card1)
        hand1.remove(card1)
        self.assertEqual(hand1.cardList == cardList, True)

    def testSort(self):
        card1 = Card(9, 1)
        card2 = Card(5, 0)
        card3 = Card(3, 0)
        cardList = []
        cardList.append(card1)
        cardList.append(card2)
        cardList.append(card3)
        hand1 = Hand(cardList)
        hand1.sort()
        cardList = []
        cardList.append(card3)
        cardList.append(card2)
        cardList.append(card1)
        for i, card in enumerate(cardList):
            self.assertEqual(hand1.cardList[i].equalTo(card),
                True)

if __name__ == '__main__':
    unittest.main()
```

```python
import unittest
from Card import Card
from Deck import Deck

class TestCards(unittest.TestCase):

    def testContains(self):
        deck1 = Deck()
        card1 = Card(14, 0)
        self.assertEqual(deck1.contains(card1), True)

    def testRemoveCard(self):
        deck1 = Deck()
        card1 = Card(14, 0)
        deck1.removeCard(card1)
        self.assertEqual(deck1.contains(card1), False)

    def testMakeHand(self):
        deck1 = Deck()
        hand1 = deck1.makeHand(5)
        self.assertEqual(len(hand1) == 5, True)
        self.assertEqual(len(deck1.getCards()) == 47, True)

if __name__ == '__main__':
    unittest.main()
```

```python
import unittest
from Card import Card

class TestCards(unittest.TestCase):

    def testEqualTo(self):
        card1 = Card(3, 0)
        card2 = Card(3, 0)
        self.assertEqual(card1.equalTo(card2), True)
        card3 = Card(3, 1)
        self.assertEqual(card1.equalTo(card3), False)

    def testBeats(self):
        card1 = Card(3, 0)
        card2 = Card(5, 0)
        trump = 0
        leadSuit = 1
        self.assertEqual(card2.beats(card1, leadSuit, trump),
            True)
        card3 = Card(10, 1)
        self.assertEqual(card1.beats(card3, leadSuit, trump),
            True)
        card4 = Card(14, 1)
        self.assertEqual(card3.beats(card4, leadSuit, trump),
            False)
        card5 = Card(14, 0)
        self.assertEqual(card5.beats(card4, leadSuit, trump),
            True)

if __name__ == '__main__':
    unittest.main()
```