University of Manchester
School of Computer Science
Project Report 2018

Learning for Hidden Information Games

Author: Pranav Bahuguna

Supervisor: Dr. Jon Shapiro

Learning for Hidden Information Games

Author: Pranav Bahuguna

In recent times, the usage of machine learning techniques to enable computers to make smarter decisions has grown exponentially. One area of particular interest for such techniques is the field of game-playing, especially as advances in processing power have allowed the analysis of complex games to be possible.

This project will focus on a particular class of games that are non-deterministic and contain hidden information. As solving such games directly can be prohibitively difficult, I will focus on finding an approximate solution (an ϵ-Nash equilibrium) using the Counterfactual Regret Minimization algorithm (CFRM).

My implementation was written in C++ and Python in Microsoft's Visual Studio. It was able to discover an ϵ-Nash equilibrium for both 2 and 3-player Kuhn Poker where $\epsilon < 0.01$. Additionally, I was able to reduce the average calculation time for 10 million iterations of 3-player Kuhn Poker from 193.2 seconds to 64.2 seconds after several optimisations.

There is still much room for improvement in this project. I have not analysed games more complex than 3-player Kuhn Poker with this algorithm and there are many other potential optimisations that could be investigated to determine if they provide any benefit.

Supervisor: Dr. Jon Shapiro

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project Overview

The primary aim of this project is to implement machine learning algorithms to develop programs that can effectively play games with hidden information. My intention here is to create such programs to handle games of increasing complexity, starting with Rock-Paper-Scissors, then 2 and 3-player Kuhn Poker. These programs will achieve these goals by finding a strategy set for each game in which a Nash equilibrium can be found. As usage of machine learning methods will only provide an approximation, finding an ϵ-Nash equilibrium where $\epsilon < 0.01$ will be considered acceptable.

The secondary aim of this project is to find and implement any methods of optimising the machine learning algorithms used. Since these algorithms must be run for many iterations to find a close ϵ-Nash equilibrium, this process can take a significant amount of time. Thus discovering ways to speed up the underlying algorithms will be highly beneficial.

1.2 Motivation

There are several motivations I have for pursuing this particular project. One major reason is that using machine learning techniques for determining the optimal strategy in games can reveal interesting data regarding a particular game. For example, one can determine if one player has an inherent advantage in that game by following the optimal and how large their expected win/loss rate is.

Another motivation is to demonstrate that we can use such techniques to empirically discover an ϵ-Nash equilibrium instead of mathematically determining one (which can be extremely difficult for more complex games). One can then use the same principles to solve more complicated games.[1] We can also compare different approaches using the same techniques to discover which ones are the fastest at solving different games.

1.3 Timeline

In order to maintain a consistent pace of development, the project was split into a number of achievable milestones. The list of goals is as follows:

- Research and understand regret minimization concepts and algorithms
- Develop designs for implementing Regret Minimization with Rock-Paper-Scissors
- Implement Rock-Paper-Scissors using Regret Minimization
- Develop tools for data visualisation and analysis of results
- Research the Counterfactual Regret Minimization (CFRM) algorithm for Kuhn Poker
- Develop designs for implemnting CFRM with 2 and 3-player Kuhn Poker
- Implement 2-player Kuhn Poker using CFRM
- Implement 3-player Kuhn Poker using CFRM
- Research any methods of optimising CFRM for both Kuhn Poker variants
- Design and implement any optimisations and compare the results

[1]Martin Zinkevi and Piccione, *Regret Minimization in Games with Incomplete Information*, pp. 1–2.

2 Background

2.1 Game Definition

The games being dealt with in this project are what are known as **hidden-information** games. This property arises from that each player does not know the exact game state that another player is in. The player thus cannot make an informed decision using this information as it is *hidden* from them. For example, the action that each player chooses in Rock-Paper-Scissors is not known to the other player until the actual showdown. In the Poker game variants, the cards dealt to each player are also hidden. Thus when one player takes an action, the other players do not know what context that player made that action under.

Since the players lack game state information, they can instead try to guess what state another player is in. This introduces a factor of chance in such strategies, and so these games are considered **non-deterministic**. This property differs these games from **deterministic** games such as Chess and Go, which lack a chance element. Any learning algorithm we use for non-deterministic games must be able to account for this chance factor. Some games such as backgammon are non-deterministic, but they lack the hidden-information property present in games like Poker and thus will not be the focus of this project.

To be able to easily describe games in a consistent manner, we must define the following terms first:

- **Strategy**: When it is a certain player's turn in a game, that player has access to a set of actions that can be taken. A *pure strategy* is a single one of these actions being played with probability 1, while a *mixed strategy* is a probability distribution over all pure strategies.
- **Utility**: The utility of an action is the expected player reward payoff for having chosen that action. This ŕewardćan be both positive or negative.

- **Information Set**: This is a set of all states belonging to one player that an opponent cannot distinguish between due to lacking the necessary information. They are thus treated as the same node. In Kuhn Poker, this means that the decision node for a player while having a Jack, Queen or King respectively are treated by an opponent as the same, since the opponent does not know what card the other player is holding at that information set.

Typically in most hidden-information games, a player will opt to use a mixed strategy. This is since usage of a pure strategy can easily be exploited by an opponent. For example in RPS, if one player uses a pure strategy of choosing rock with 100% probability, the opponent can exploit this by using a pure strategy of choosing paper every time. If using a mixed strategy however, it is much more difficult for an opponent to predict future actions based on previous ones, thus reducing the exploitability of a player's chosen strategy considerably.

2.2 Nash Equilibrium

A **Nash Equilibrium** is defined as a set of strategies for each player such that each player cannot improve their final game payoffs by changing their strategy. Each player is expected to know the equilibrium strategies of the other players in this scenario. It is named after the American mathematician John Forbes Nash Jr. The goal of this project is to discover a Nash equilibrium for our selected games.

Why are we interested in finding the Nash equilibrium? As no player's final payoff improves from changing their strategies corresponding to a Nash equilibrium, these strategies are thus considered optimal for each player. For simple games, it is possible to mathematically determine the Nash equilibrium strategies. Using machine learning techniques, we can only reach an approximation instead. We thus search instead for an ϵ-Nash equilibrium, where ϵ is the largest difference in distance of obtained utilities from expected utilities.

2.3 Example Games

2.3.1 Rock-Paper-Scissors

In Rock-Paper-Scissors (RPS), there are two players who can each play one of three different actions (rock, paper, scissors). It is a **one-shot** game, meaning that all players choose a single action taking place simultaneously in a single turn. It is also considered to be **zero-sum** game as either player's gain in payoffs is the other player's loss.

To better represent this game mathematically, it can be defined as a **normal-form** game. A normal-form game can be represented as an n-dimensional table, where each dimension has rows/columns corresponding to one player's actions, and each table entry corresponds to a single action profile (combination of each player's chosen action) and contains a vector of utilities for each player. The utility matrix for RPS is as follows:

		Player 2		
		R	P	S
	R	0, 0	-1, 1	1, -1
Player 1	P	1, -1	0, 0	-1, 1
	S	-1, 1	1, -1	0, 0

Table 2.1: Rock-Paper-Scissors utility matrix

As we can see in Table 2.1, the zero-sum property arises from the fact that the utility values for each player in each action profile sum to zero.

2.3.2 2-Player Kuhn Poker

Kuhn Poker is a simplified version of Poker developed by Harold W. Kuhn.[1] Kuhn Poker differs from RPS in that it is not a one-shot game. Rather, it takes place over several stages where each player has a turn to take an action. Like RPS however, it is zero-sum as one player's gain is anothers loss.

In the 2-player version there exists three cards (Jack, Queen, King), two of which are dealt to each player. There is thus 6 different ways of dealing one card to each player. At each

[1]Kuhn and Tucker, *Contributions to the Theory of Games*, pp. 97–103.

decision node, a player has two separate actions that they can take (pass or bet). At each terminal node, one of the players either automatically wins the pot, or there is a showdown for the pot where the player with the highest card wins. A game of 2-player Kuhn Poker proceeds as follows:

1. Each player antes 1.
2. Each player is randomly dealt one of three cards, the third is put away unseen.
3. Player 1 can pass or bet 1.
    - 3.1. If Player 1 passes, Player 2 can pass or bet.
        - 3.1.1. If Player 2 passes, there is a showdown for a pot of 2.
        - 3.1.2. If Player 2 bets, Player 1 can pass or bet 1.
            - 3.1.2.1. If Player 1 passes, Player 2 wins the pot of 3.
            - 3.1.2.2. If Player 1 bets, there is a showdown for a pot of 4.
    - 3.2. If Player 1 bets, Player 2 can pass or bet.
        - 3.2.1. If Player 2 passes, Player 1 wins the pot of 3.
        - 3.2.2. If Player 2 bets, there is a showdown for a pot of 4.

Due to the multiple stages involved, this is best defined as an **extensive-form** game. This form represents the game as a tree and displays several key features including:

- Decision and terminal nodes
- Possible player actions from each decision node
- Information each player has about other players moves at each stage
- Terminal utility payoffs for each possible game outcome
- Representation of chance events

An extensive-form representation of 2-Player Kuhn Poker is shown in Figure 2.1:

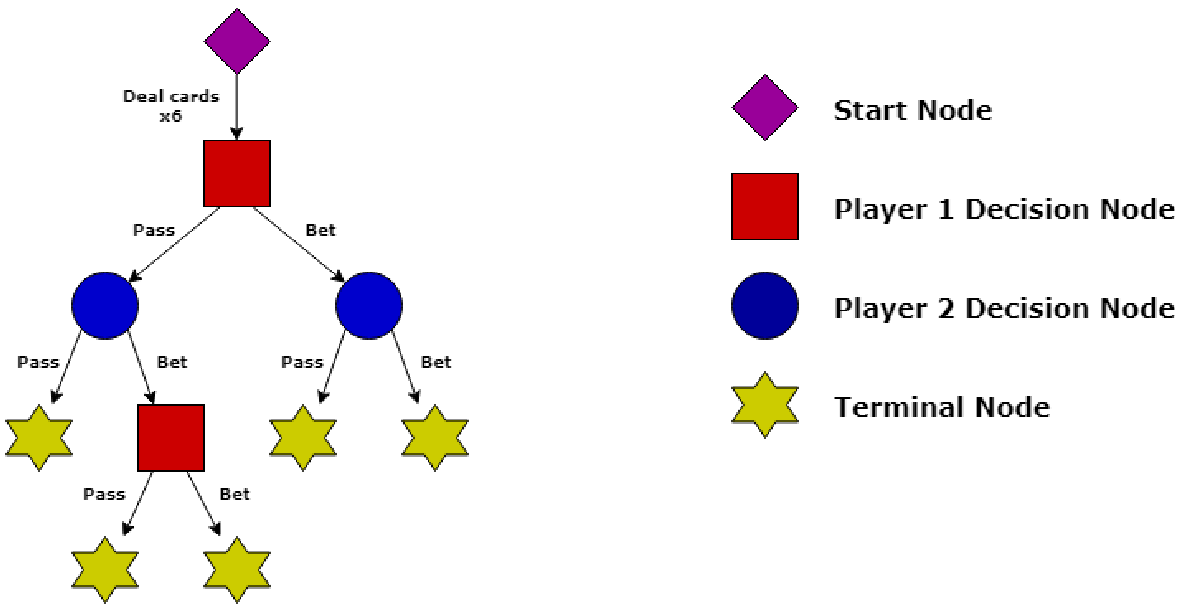


Figure 2.1: 2-Player Kuhn Poker Game Tree

2.3.3 3-Player Kuhn Poker

The 3-player version has several noticeable differences. There are now 4 cards (Jack, Queen, King, Ace) available to be dealt from. This provides 24 different ways of dealing one card to each player. This game tree is also significantly larger, with 12 decision and 13 terminal nodes.

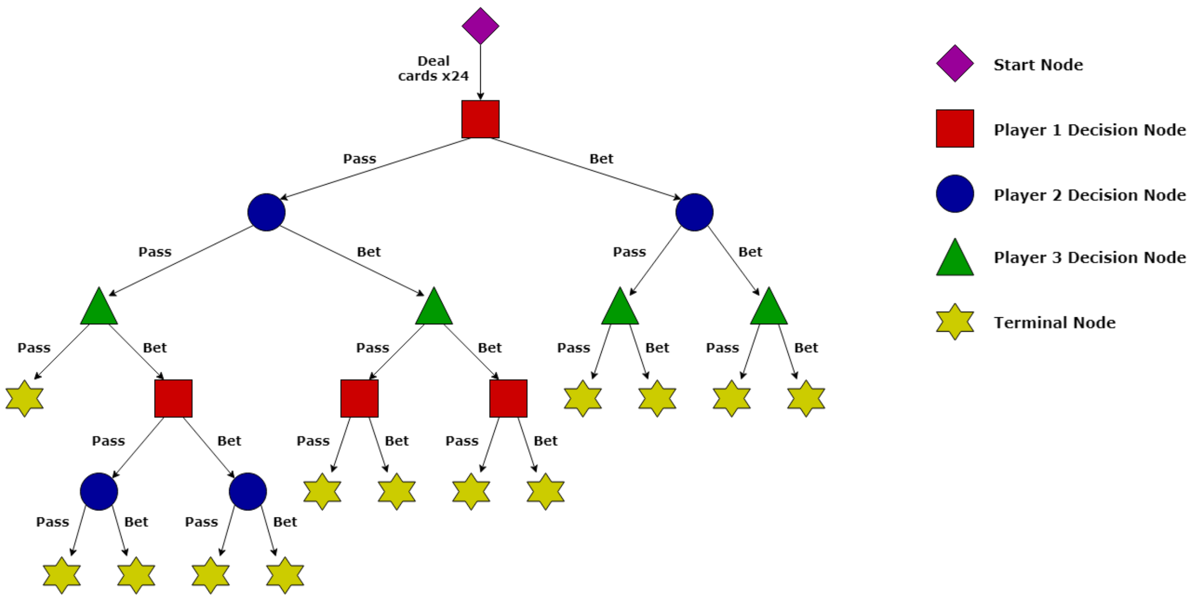


Figure 2.2: 3-Player Kuhn Poker Game Tree

3 Reinforcement Learning

3.1 What is Reinforcement Learning?

Reinforcement learning is an area of machine learning that frequently sees use in game theory research. It is primarily concerned with how software agents should take actions in a certain environment to maximise some cumulative reward. There are several major components involved in a reinforcement learning algorithm:[1]

- **Agent**: This is the entity that is actively making use of the learning algorithm. In our case, the game players are agents.
- **Environment**: The context and reward system which the algorithm is running under.
- **States**: The environment can be composed of several states, one of which an agent may be currently operating under. An agent may have to adjust their decision-making process depending on the state they are in. These are analogous to the decision nodes that exist in multi-stage games like Kuhn Poker.
- **Reward-Function**: This is some incentive an agent receives for reaching a certain state. In the context of game playing, this refers to the payoff given to a player for performing an action at a state.
- **Value-Function**: This function enables an agent to determine the total cumulative reward they can obtained from a state and all subsequent ones. The agent can use this information to prioritise reaching states with a greater value to maximise rewards over many games, rather than just the most immediate one.
- **Policy**: The agent's policy defines behaviour at a specific state. It encompasses how the agent makes use of the available information and value function to make decisions.

[1]Skymind, *A Beginner's Guide to Deep Reinforcement Learning*.

There are other machine learning approaches that are available including supervised and unsupervised learning. There are several key properties reinforcement learning has that make it better suited for game-playing however:

- **Lack of supervision**: There is no external supervisor providing feedback on whether the agent's strategy is any good, only the agent can determine this. This is especially useful for more complicated games in which a Nash equilibrium is prohibitively difficult to mathematically determine.
- **Concept of rewards**: Unlike unsupervised learning, reinforcement learning adds a concept of reward for reaching a certain state. Instead of simply finding similarities between inputs and outputs, we find what the algorithm considers to be 'good behaviour'.
- **Inclusion of temporal concepts**: Reinforcement learning pays attention to which state an agent is in before making a decision. It can also account for cases in which a reward is not granted immediately, but after completion of several other steps. This is particularly useful for games like Kuhn Poker - what may seem advantageous at an early stage may not be so at a later one.

3.2 Regret Minimization

If we look back at our game example for RPS, we find that a typical game proceeds as follows:

1. Two players select one of three available actions.
2. Both players simultaneously reveal their chosen action. Each player calculates their respective utilities and starts a new game.

To compute the *expected utility* of the game for an agent, we can sum over each action profile the product of each player's probability of player their specific action, multiplied by the player's utility for the action profile. This can be represented mathematically (as shown in 3.1) by making the following definitions:[2]

- Let S_i be a finite set of actions available for player i

[2]Neller and Lanctot, *An Introduction to Counterfactual Regret Minimization*, pp. 2–3.

- Let σ to refer to a mixed strategy and $\sigma_i(s)$ be the probability of player i choosing action $s \in S$.
- Let u_i be a vector of utilities for player i.
- We also define $-i$ to refer to player i's opponents. Thus in a two player game if $i = 1$, then $-i = 2$.

$$u_i(\sigma_i, \sigma_{-i}) = \sum_{s \in S_i} \sum_{s' \in S_{-1}} \sigma_i(s)\sigma_{-i}(s')u_i(s, s') \tag{3.1}$$

How should each player choose their action for the next game? One way of doing so is to select the action the player *regrets* not having chosen before. Suppose player 1 chooses rock and player 2 chooses paper. Player 1 would regret not having chosen scissors before, so it would be useful to choose scissors next time in expectation of player 2 choosing paper again.

We can calculate the *regret* of not playing an action by subtracting the utility of the action we actually played from the utility of that action. For our above example, we can calculate the following regrets:

$$Regret(R) = Utility(R, P) - Utility(R, P) = -1 - (-1) = 0 \tag{3.2}$$
$$Regret(P) = Utility(P, P) - Utility(R, P) = 0 - (-1) = 1 \tag{3.3}$$
$$Regret(S) = Utility(S, P) - Utility(R, P) = 1 - (-1) = 2 \tag{3.4}$$

As we can see above, player 1 would regret not having played paper as they could have achieved a draw, but would regret not having played scissors *even more* as doing so would guarantee a win.

However, simply choosing the action one regretted not playing the most is not a good idea. This is since the opponent can also discover the regretted non-played action and react accordingly (e.g. play rock). What one can instead do is to utilise a mixed strategy in which possible actions are randomly selected with a probability proportional to the positive regrets. This process, known as **regret matching** allows us to reduce our exploitability when constructing a new strategy by maximising our expected utilities.

Over many games, we would like to reduce minimize our expected regrets over time. To do this, we perform regret matching with *cumulative regrets* rather than our most immediate ones. From this, we can construct an algorithm for performing regret minimization over many games:

1. For each player, initialize all cumulative regrets to 0.
2. For a number of iterations, perform the following:
    - 2.1. Compute a regret-matching strategy (if all regrets are non-positive, use a uniform random strategy).
    - 2.2. Add the strategy to the strategy profile sum.
    - 2.3. Select each player action according to the computed strategy.
    - 2.4. Play game and compute resultant regrets.
    - 2.5. Add regrets to cumulative regrets.
3. Return the average strategy used - the strategy profile sum divided by the number of iterations.

By repeating this process over many iterations, we should find that the average strategy for each player converges towards a Nash equlibrium. In the case of RPS, this should result in a mixed strategy of $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ for both players.

3.3 Counterfactual Regret Minimization

The CFRM algorithm is an extension of the regret minimization algorithm we saw before. Unlike regret minimization, it deals with games with multiple stages and so must account for the following factors:

- The probability of reaching a state given the players' strategies.
- How to pass forward game state information and probabilities of player actions, and pass back utility information through these states.

The CFRM algorithm can be mathematically represented by defining the following:[3]

- Let A denote the set of all game actions, I denote an information set and $A(I)$ denote the set of legal actions at information set I.
- Let t denote time steps (increment with resepect to each information set) and T represent the total number of game iterations.
- A strategy σ_i^t thus maps player i's information set I_i and action $a \in A(I_i)$ to the probability of choosing a in I_i at time t. We also let $\sigma_{I \to a}$ denote a profile equivalent to σ, except action a is always chosen at information set I.
- A history h represents a sequence of actions (including chance) starting from game root.
- Let $\pi^\sigma(h)$ represent the probability of reaching history h with strategy profile σ. In addition, let $\pi^\sigma(I)$ be the probability of reaching I through all possible histories in I, i.e. $\pi^\sigma(I) = \sum_{h \in I} \pi^\sigma(h)$. Finally, let $\pi_{-i}^\sigma(I)$ be the probability of reaching I using σ except we treat player i's actions to reach the state as having probability 1. This is known as the *counterfactual reach probability.* When we refer to ćounterfactual; we treat the computation as if player i's strategy was changed to intentionally reach information set I_i.
- Let Z be the set of all terminal histories (root to leaf), then prefix $h \sqsubset z$ for $z \in Z$ is a non-terminal history.

The *counterfactual value* at non-terminal history h is defined as:

$$v_i(\sigma, h) = \sum_{z \in Z, h \sqsubset z} \pi_{-i}^\sigma(h) \pi^\sigma(h, z) u_i(z) \tag{3.5}$$

The *counterfactual regret* of not taking action a at history h is:

$$r(h, a) = v_i(\sigma_{I \to a}, h) - v_i(\sigma, h) \tag{3.6}$$

[3]Martin Zinkevi and Piccione, *Regret Minimization in Games with Incomplete Information*, pp. 2–3; Neller and Lanctot, *An Introduction to Counterfactual Regret Minimization*, pp. 10–11.

Thus the *counterfactual regret* of not taking action a at history h is:

$$r(I,a) = \sum_{h \in I}(h,a) \tag{3.7}$$

For a specific time step t and player i, the above can be more specifically given as $r_i^t(I,a)$. The cumulative counterfactual regret is then defined as:

$$R_i^T(I,a) = \sum_{t=1}^{T} r_i^t(I,a) \tag{3.8}$$

Finally by defining the non-negative counterfactual regret as $R_i^{T,+}(I,a) = max(R_i^T(I,a),0)$, the regret matching algorithm can be applied to obtain the full algorithm for counterfactual regret minimization:

$$\sigma_i^{T+1}(I,a) = \begin{cases} \frac{R_i^{T,+}(I,a)}{\sum_{a \in A(I)} R_i^{T,+}(I,a)} & \text{if } \sum_{a \in A(I)} R_i^{T,+}(I,a) > 0 \\ \frac{1}{|A(I)|} & \text{otherwise} \end{cases} \tag{3.9}$$

Despite the apparent complexity of the equation, the CFRM algorithm is not overly difficult to implement in code. A programatic implementation of the algorithm proceeds as follows for a single iteration:

1. Obtain the strategy for the player whose turn it is at this node.
2. Follow each possible action from this node and pass forward the probability of the current player reaching that child node (the reach probability or *weight*).
   2.1. If we reach a decision node, we repeat the above steps again.
   2.2. If we reach a terminal node, we calculate the terminal utilities and pass back this information to the parent decision node.
3. The parent decision node calculates their own utilities by adding the child utilities multiplied with the probability of taking the action leading to that node.
4. We then calculate the *counterfactual reach probability* of this node. By ćounterfactual; we treat the computation as if the current players' strategy was adjusted to have inten-

tionally played to reach this current node. We calculate this by multiplying the weights of all players at this node except the current one.

5. We then update the cumulative regrets by adding the regrets of each possible action multiplyed by the counterfactual reach probability for this node.

6. The decision node utilities are passed back to their parent node, repeating the process until the root node is reached.

Like regret minimization, repeating this algorithm for many iterations should produce an average strategy for every decision node that converges towards a Nash equilibrium.

4 Project Design

4.1 Tools and Software

Before starting any design or implementation work on game-playing programs, I found it necessary to make up-front decisions on what software and programming languages to use for building the project. I prioritised this aspect of the project as I believed the underlying software being used would have the greatest impact and thus clear decisions regarding this needed to be made first.

I ultimately chose to write the main project infrastructure in C++ using Microsoft Visual Studio. The usage of such tools and languages provided several advantages I felt were useful for my project:

- C++ is multi-paradigm and supports object-oriented programming. As I opted to take an object-oriented approach to program design, this feature was essential.
- The machine learning algorithms involved often had to be run millions of times to achieve accurate results. This would make the program very resource intensive, so good performance was a major consideration. C++ generally allows for faster and more efficient code to be written as well as more precise control over memory allocation / deallocation than equivalent code in Java or Python.
- C++ has access to a wide range of 3rd party libraries to perform useful actions or significantly reduce the amount of time and effort spent in building the project.
- The Visual Studio IDE provides many useful functions for building projects, including but not limited to reference finding, code completion, testing and debugging facilities, performance profiling etc.

4.1.1 Data Visualisation

Another important aspect of the project was to produce visible results from the learning process that can be analysed and allow us to draw conclusions from. Desired results include game values for each player, the strategies used, the ϵ-Nash equilibrium and graphs of all the previous properties over time.

While the result collection and manipulation was done in C++, I opted to write the graphing facilities in Python using Matplotlib. I did this since C++ lacked any graphing libraries I found satisfactory, whereas Python provided an excellent graphing library that was well supported and simple to use. I was able to to integrate calls to Python functions from the main C++ program, allowing graphs of the processed results to be displayed immediately after completion of the learning algorithm.

4.2 Game Design

For each of the games I designed this project to work with, the approaches used varied quite significantly. Part of the reason for this was due to the differences between the games in question (especially between RPS and Kuhn Poker variants). Another major reason however was that I obtained better insights into improving the structural design of games from the experience gained in previous attempts.

In spite of the differences between the implementations for the various games, they all contain three major structures to support their operation:

- **Main**: This structure is responsible for collecting and applying user options, setting up and running games and collecting and displaying results. It represents the main centre of control as well as the first point of entry into the program.
- **Game**: This structure defines the rules, states, rewards and the progression of a specific game. It also contains methods for collecting various statistics from running the game over many iterations.
- **AI**: This structure implements the learning algorithm used by each player to construct a strategy for game-playing, as well as methods for updating their policy based on game outcomes.

These structures are not rigidly defined and are often partially merged or split over several different classes. They do however capture the main functionalities that are common to all implementations.

4.3 Rock-Paper-Scissors

In RPS, I had chosen to keep the learning and decision-making functionalities within a *Player* class. Both players in the game are represented by a *Player* object, so their respective decision-making policies and strategies are kept separately. The actual *Game* class is quite lightweight and simply runs the strategy calculation and training process for a number of iterations. As described before, the *Main* class handles the main program settings and result collection. It also holds a *GraphPlotter* instance for displaying the collected results, which feeds its calls to a Python script that calls the actual Matplotlib functions for drawing graphs.

The UML diagram below illustrates the main class structure used in the RPS trainer project. I have however omitted several attributes and functions that I believe were unimportant for accurately describing the class structure.

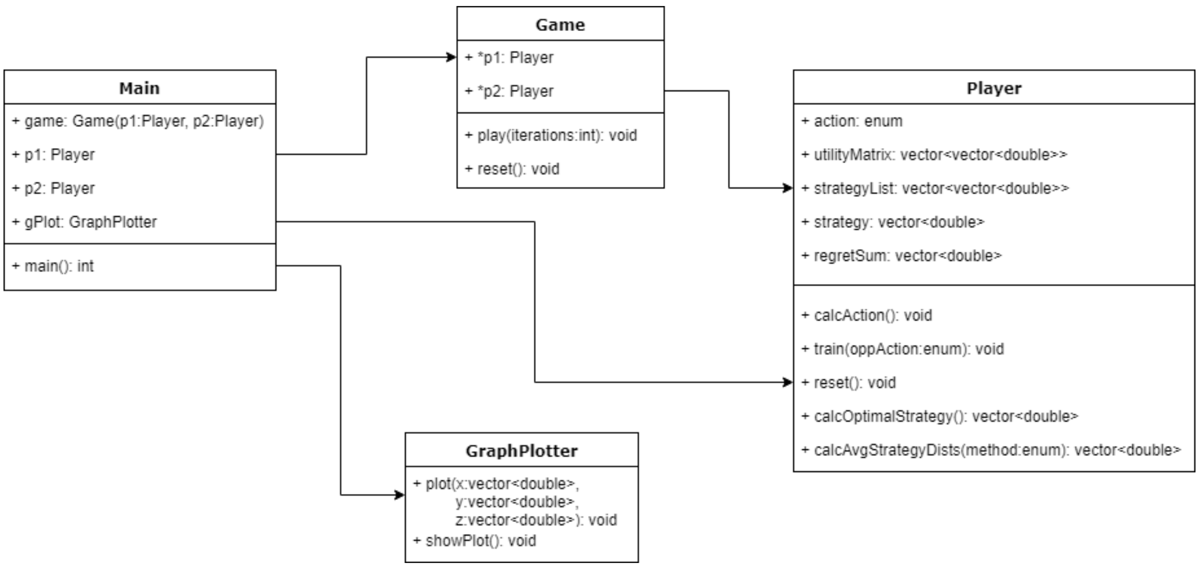

Game
+ *p1: Player
+ *p2: Player
+ play(iterations:int): void
+ reset(): void
Main
+ game: Game(p1:Player, p2:Player)
+ p1: Player
+ p2: Player
+ gPlot: GraphPlotter
+ main(): int
Player
+ action: enum
+ utilityMatrix: vector<vector<double>>
+ strategyList: vector<vector<double>>
+ strategy: vector<double>
+ regretSum: vector<double>
+ calcAction(): void
+ train(oppAction:enum): void
+ reset(): void
+ calcOptimalStrategy(): vector<double>
+ calcAvgStrategyDists(method:enum): vector<double>
GraphPlotter
+ plot(x:vector<double>, y:vector<double>, z:vector<double>): void
+ showPlot(): void

Figure 4.1: UML class diagram for Rock-Paper-Scissors

4.4 Kuhn Poker

My design for Kuhn Poker was intended to work for both 2 and 3-player versions. The intention of this was to minimise the amount of duplicated effort as most of the components used by each version are highly similar. The differences between the two only affect the game tree and number of players, both of which the *Game* class can account for.

The concept of a specific *Player* class has been discarded, being replaced by the *DecisionNode* class. Each *DecisionNode* object represents a game tree state in which the current player can decide what action to take. It is in here where the AI facilities previously provided by the *Player* class are now utilised. Another class is *TerminalNode*, which represent the game tree leaf (end) states and store the list of players still contesting at them. If more than one player is contesting, then only a card showdown between contestants can decide a winner.

The scope of the *Game* class has greatly expanded. It now performs the CFRM algorithm in which every applicable *DecisionNode* object can update their respective strategies. The *Game* class also takes responsibility from the *Main* class for data wrangling, while the latter performs the same role as before in program control and result presentation via *GraphPlotter*.

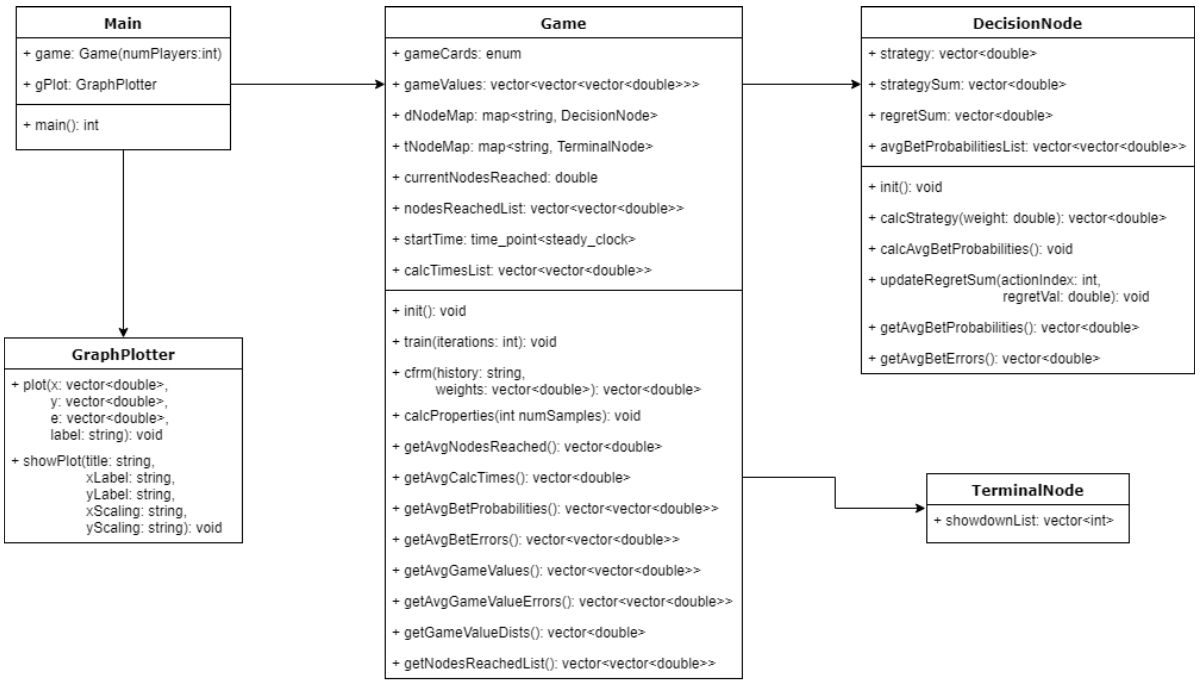

Main
+ game: Game(numPlayers:int)
+ gPlot: GraphPlotter
+ main(): int
GraphPlotter
+ plot(x: vector<double>, y: vector<double>, e: vector<double>, label: string): void
+ showPlot(title: string, xLabel: string, yLabel: string, xScaling: string, yScaling: string): void
Game
+ gameCards: enum
+ gameValues: vector<vector<vector<double>>>
+ dNodeMap: map<string, DecisionNode>
+ tNodeMap: map<string, TerminalNode>
+ currentNodesReached: double
+ nodesReachedList: vector<vector<double>>
+ startTime: time_point<steady_clock>
+ calcTimesList: vector<vector<double>>
+ init(): void
+ train(iterations: int): void
+ cfrm(history: string, weights: vector<double>): vector<double>
+ calcProperties(int numSamples): void
+ getAvgNodesReached(): vector<double>
+ getAvgCalcTimes(): vector<double>
+ getAvgBetProbabilities(): vector<vector<double>>
+ getAvgBetErrors(): vector<vector<double>>
+ getAvgGameValues(): vector<vector<double>>
+ getAvgGameValueErrors(): vector<vector<double>>
+ getGameValueDists(): vector<double>
+ getNodesReachedList(): vector<vector<double>>
DecisionNode
+ strategy: vector<double>
+ strategySum: vector<double>
+ regretSum: vector<double>
+ avgBetProbabilitiesList: vector<vector<double>>
+ init(): void
+ calcStrategy(weight: double): vector<double>
+ calcAvgBetProbabilities(): void
+ updateRegretSum(actionIndex: int, regretVal: double): void
+ getAvgBetProbabilities(): vector<double>
+ getAvgBetErrors(): vector<double>
TerminalNode
+ showdownList: vector<int>

Figure 4.2: UML class diagram for Kuhn Poker

5 Testing

5.1 Testing Methodology

While one can build a program that claims to implement a machine learning algorithm for a specific game, it may not necessarily function correctly. It is thus necessary to check whether the results match what is expected of the algorithm.

The games I am implementing these algorithms with are considered *solved*. This means the optimal strategies for those games leading to a Nash equilibrium have already been mathematically determined:

- **RPS**: The Nash strategies are obtained by calculating the normalised kernel of the utility matrix for each player. Assuming we are using the utility matrix described in Table 2.1, we should obtain an optimal strategy of $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ for both players.
- **Kuhn Poker**: 2-player Kuhn Poker was solved by Harold W. Kuhn who had developed the game in 1950. The 3-player variant however was not solved until recently in 2013 by the work of Szafron, Gibson and Sturtevant.[1]

The solutions for 2 and 3-player Kuhn Poker are given in Table 5.2 and Table 5.3. We refer to each decision node as the 1st - 4th node with a specific card. They are numbered according to the order in which that node can be reached by a specific player traversing the game tree in depth-first order. Table 5.1 illustrate which decision nodes belong to each player for 2 and 3-player Kuhn Poker, identifying them by the history of actions required to reach each one.

Also note for 2-player Kuhn Poker, we have *free variable* α, which can hold any value provided it is consistent in all equations. For 3-player Kuhn Poker we have β and γ, where $\beta = max(B_{J1}, B_{Q1})$ and $\gamma = \frac{1}{24}$.

[1]Duane Szafron, *A Parameterized Family of Equilibrium Profiles for Three-Player Kuhn Poker*.

	Player 1	Player 2
1st node	–	p
2nd node	pb	b

	Player 1	Player 2	Player 3
1st node	–	p	pp
2nd node	ppb	ppbp	pb
3rd node	pbp	ppbb	bp
4th node	pbb	b	bb

Table 5.1: 2 and 3-player Kuhn Poker node histories

	Player 1	Player 2
Bet at 1st node with J	$A_{J1} = 0$	$B_{J1} = \frac{1}{3}$
Bet at 2nd node with J	$A_{J2} = 0$	$B_{J2} = 0$
Bet at 1st node with Q	$A_{Q1} = 0$	$B_{Q1} = \frac{1}{3}$
Bet at 2nd node with Q	$A_{Q2} = \alpha + \frac{1}{3}$	$B_{Q2} = \frac{1}{3}$
Bet at 1st node with K	$A_{K1} = 3\alpha$	$B_{K1} = 1$
Bet at 2nd node with K	$A_{K2} = 1$	$B_{K2} = 1$
Game Values	$u_1 = -\frac{1}{18}$	$u_2 = \frac{1}{18}$

Table 5.2: 2-Player Kuhn Poker Optimal Strategies

	Player 1	Player 2	Player 3
Bet at 1st node with J	$A_{J1} = 0$	$B_{J1} \leq B_{Q1}$ if $C_{J1} = 0$	$C_{J1} \leq min(\frac{1}{2}, \frac{2-B_{J1}}{3+2B_{J1}+2B_{Q1}})$
		$B_{J1} \leq \frac{1}{4}$ otherwise	
Bet at 1st node with Q	$A_{Q1} = 0$	$B_{Q1} \leq \frac{1}{4}$ if $C_{J1} = 0$	$C_{Q1} = \frac{1}{2} - C_{J1}$
		$B_{Q1} = B_{J1}$ if $0 < C_{J1} < \frac{1}{2}$	
		$B_{Q1} \leq min(B_{J1}, \frac{1}{2}(1-B_{J1}))$	
Bet at 1st node with K	$A_{K1} = 0$	$B_{K1} = 0$	$C_{K1} = 0$
Bet at 1st node with A	$A_{A1} = 0$	$B_{A1} = 0$	$C_{A1} = 1$
Bet at 2nd node with J	$A_{J2} = 0$	$B_{J2} = 0$	$C_{J2} = 0$
Bet at 2nd node with Q	$A_{Q2} = 0$	$B_{Q2} \leq max(0, \frac{B_{J1}-B_{Q1}}{2(1-B_{Q1})})$	$C_{Q2} = 0$
Bet at 2nd node with K	$A_{K2} = 0$	$B_{K2} = \frac{1}{2}(1 + B_{J1} + B_{Q1} + \beta) - B_{Q2}(1 - B_{Q1})$	$C_{K2} = 0$
Bet at 2nd node with A	$A_{A2} = 1$	$B_{A2} = 1$	$C_{A2} = 1$
Bet at 3rd node with J	$A_{J3} = 0$	$B_{J3} = 0$	$C_{J3} = 0$
Bet at 3rd node with Q	$A_{Q3} = 0$	$B_{Q3} \leq max(0, \frac{B_{J1}-B_{J4}}{2(1-B_{Q1})})$	$C_{Q3} = 0$
Bet at 3rd node with K	$A_{K3} = \frac{1}{2}$	$B_{K3} = 0$	$\frac{1}{2} - B_{K3} < C_{J4} < \frac{1}{4}(2 + 3(B_{J1} + B_{Q1})) - B_{K4}$
Bet at 3rd node with A	$A_{A3} = 1$	$B_{A3} = 1$	$C_{A3} = 1$
Bet at 4th node with J	$A_{J4} = 0$	$B_{J4} = 0$	$C_{J4} = 0$
Bet at 4th node with J	$A_{Q4} = 0$	$B_{Q4} = 0$	$C_{Q4} = 0$
Bet at 4th node with J	$A_{K4} = 0$	$B_{K4} \leq \frac{1}{4}(2+3(B_{J1}+B_{Q1})+\beta)$	$0 \leq C_{K4} \leq 1$
Bet at 4th node with J	$A_{A4} = 1$	$B_{A4} = 1$	$C_{A4} = 1$
Game Values	$u_1 = -\gamma(\frac{1}{2} + \beta)$	$u_2 = -\gamma(\frac{1}{2})$	$u_3 = \gamma(1 + \beta)$

Table 5.3: 3-Player Kuhn Poker Optimal Strategies

5.2 Test Design

Using the equations given for describing the optimal strategies for each player, we can construct tests to compare the algorithm outputs to what we expect to receive. As the output strategy values are not expected to exactly match the expected values, we instead check if they are within a certain tolerance.

I implemented these tests as unit tests using the Google Test framework in C++. For both Kuhn Poker variants, I ran the games for 10 million iterations before testing the outputs. I chose this number as I considered it a good tradeoff between output accuracy and time taken to run the tests. As I could expect to run these tests quite frequently throughout the project, it was important that I could quickly obtain results. I also chose to use tolerances of 0.01 and 0.03 for 2 and 3-player Kuhn Poker respectively. This was another trade-off in confirming the accuracy of the results (distance of generated strategy from what was expected) without generating too many false negatives from failing tests.

The tests are included in the project and can be run separately from the main code. When run, they will run both games for 10 million iterations each, then measure the distance of each strategy variable from the expected result. Any variables that exceed the tolerance will be cause that particular test to fail and record the result. This allows failing tests to be quickly identified.

6 Optimisation

After sucessfully implementing the CFRM algorithm for 2 and 3-player Kuhn Poker, I focused on finding ways to optimise the algorithm. Currently, the basic recursive implementation took an average of 193 seconds to run the CFRM algorithm on 3-player Kuhn Poker for 10 million iterations on my machine. More complicated games would need far more iterations and more time per iteration to find a close ϵ-Nash equilibrium, which could easily lead to requiring days of continuous processing. Discovering ways of speeding up the algorithm was thus something I considered important as it can potentially save significant amounts of time and resources in future studies.

There were two major approaches I used to achieve speedup. One method was to increase the amount of useful work being done per unit time, and the other was to reduce the total amount of work that was required to be done. The first approach was pursued using multithreading and performance profiling while the second approach was achieved with algorithmic improvements.

6.1 Algorithmic Improvements

One method used to reduce the algorithmic workload is **tree pruning**. In the CFRM algorithm, the utilities of a decision node is given by the sum of the child utilities multiplied by the probability of reaching that child. It thus stands that if the child reach probability is zero, then none of those child's utilities or any children in that subtree contribute to the parent node's utilities. This allows us to simply skip performing CFRM for that subtree and reduce the number of nodes that need visiting per iteration.

In the actual implementations, we check if the average probability of visiting the child falls below a certain threshold, rather than if it is zero. This is since the average reach probability for them never reaches zero, but reduces enough for that child's contribution to be negligible. The recursive implementation achieves this simply by not recursively calling the CFRM algorithm for the excluded subtree. The iterative implementation is more complicated and requires the algorithm to update and check a ‘blacklist‘ of excluded nodes that no calculations should be performed for.

6.2 Multithreading

Another path to pursuing potential performance gains was to try implementing multithreading in the program. In the CFRM algorithm, processing for each decision node on the same level can be accomplished independently from the others, as the subtrees they are dependent on are independent. This property makes them suitable for multithreading, as we can allow a separate thread to handle each decision node.

Using the work of João Reis for a similar application,[1] I discovered that an iterative version of the CFRM algorithm was required instead of the recursive one I had originally built. This is since a multithreaded implementation needs to process nodes in breadth-first rather than depth-first order as done in the recursive algorithm. In terms of class structure, there is acutally little difference to the recursive version as most changes were only made internally to the CFRM function. One change however was that terminal outcomes could now be calculated directly, so the *TerminalNode* class was removed.

Another consideration was on deciding which framework to implement multithreading with. For this, I chose to use OpenMP as it is a lightweight and easy to use library intended to work with existing code. It also provided facilities such as parallelizable for-loops, something that I found essential to my design.

The iterative version performs the CFRM algorithm in two passes. In the first pass, it iterates downwards through each level in the game tree from the root and calculates the strategies and weights for every decision node on each level. In the second pass, it iterates upwards through each level and calculates the resultant utilities for the same decision nodes on each level. In both phases, the for-loop in which processing is done for each decision node on a level is parallelized. This process is illustrated by the pseudocode shown below:

[1]Reis, *A GPU implementation of Counterfactual Regret Minimization.*

```
for $level \leftarrow 0$ to NUM_LEVELS-1 do
    nextLevel = level + 1;
    pIndex = level % NUM_PLAYERS;
    // #pragma omp parallel for num_threads(NUM_THREADS)
    for $levelIndex \leftarrow 0$ to num dNodes on level do
        Calculate dNodeIndex from level and levelIndex;
        Get dNode for the given level, dNodeIndex and pIndex;
        Calculate strategy at this dNode;
        for $action \leftarrow 0$ to NUM_ACTIONS do
            nlIndex = levelIndex * NUM_ACTIONS + action;
            if node at nextLevel and nlIndex is dNode then
                Update weights at nextLevel and nlIndex;
            end
        end
    end
end
```

Algorithm 1: Iterative Kuhn Poker Phase 1

```
for $level \leftarrow$ NUM_LEVELS - 2 to 0 do
    nextLevel = level + 1;
    pIndex = level % NUM_PLAYERS;
    // #pragma omp parallel for num_threads(NUM_THREADS)
    for $levelIndex \leftarrow 0$ to num dNodes on level do
        Calculate dNodeIndex from level and levelIndex;
        Get dNode for the given level, dNodeIndex and pIndex;
        for $action \leftarrow 0$ to NUM_ACTIONS do
            nlIndex = levelIndex * NUM_ACTIONS + action;
            Calculate utilities for each player at nextLevel and nlIndex;
        end
        for $action \leftarrow 0$ to NUM_ACTIONS do
            nlIndex = levelIndex * NUM_ACTIONS + action;
            Update regrets for this action at nextLevel and nlIndex;
        end
        Set utilities for current dNode to the calculated utilities;
    end
end
```

Algorithm 2: Iterative Kuhn Poker phase 2

6.3 Performance Profiling

After successfully implementing an iterative version of the CFRM algorithm, I turned towards profiling the code I had written to discover any inefficiencies and rectify them. To accomplish this, I made use of Visual Studio's profiling tools which were useful in allowing me to find how much time was being spent by the CPU in different parts of the code. Some of the improvements I implemented include:

- Pre-calculating terminal node utilities. Instead of determining the terminal utilities in the algorithm, I calculated the utilities beforehand and stored them in a look-up table, saving a significant amount of calculation per iteration.
- Improving cache locality by storing useful information (e.g. nodes, utilities, weights etc.) in array-based data structures rather than maps. While this increased the complexity of the logic to deal with them, it resulted in less cache misses when reading or writing to them.
- Passing objects by reference rather than value to functions whenever possible to avoid expensive copying.
- Increase reuse of data structures to avoid repeated object construction every iteration.

7 Evaluation

7.1 Rock-Paper-Scissors

The main objective I sought to achieve with RPS was to demonstrate how the Regret Minimization algorithm could be used find a close ϵ-Nash equilibrium quickly. To do this, I ran the algorithm for 1000 iterations and measured the distance of both player's average strategies from the calculated Nash strategies and averaged them over 10 separate trials. This experiment and all following ones were performed on a quad-core Intel Core i5 2500K running at a clock speed of 4.5GHz.

There were two main measures available for calculating the distance between the expected Nash strategy and the actual obtained strategy for a player. The first is the Euclidean distance (7.1), which is a measure of the absolute distance between two points in Euclidean space. The second is the Kullback-Liebler divergence (7.2), which is a measure of how one probability distribution diverges from another.[1]

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{n}^{i=1} (q_i - p_i)^2} \qquad (7.1) \qquad\qquad KL(\mathbf{p}, \mathbf{q}) = \sum_{n}^{i=1} p_i ln \frac{p_i}{q_i} \qquad (7.2)$$

I chose to use the Kullback-Liebler measure in this case as this measure captures a statistical meaning that the Euclidean measure does not. The distances being measured in these calculations are the difference in probability values, so Kullback-Liebler can show the divergence in probability distributions of what was expected (nash strategy) against what was actually obtained (player strategy).

As we can see in Figure 7.1, the average strategy for both players converges very quickly to the Nash equilibrium. This demonstrates that we can use the Regret Minimization algorithm to empirically discover a Nash equilibrium for one-shot games like RPS.

[1]Kullback, *Information Theory and Statistics*.

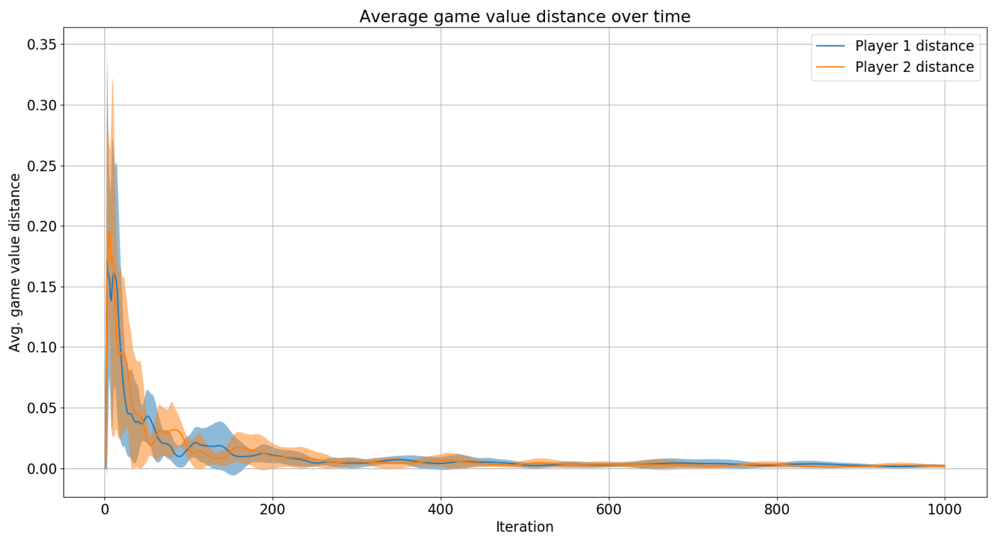


Figure 7.1: Rock-Paper-Scissors results

7.2 Kuhn Poker

For 2 and 3-player Kuhn Poker, I had more objectives that I sought to achieve. Like with RPS, I wanted to demonstrate how the CFRM algorithm can be used to find a close ϵ-Nash quickly. I also sought to compare and contrast how the various optimisations made have affected each version of the game.

For both games, I ran the CFRM algorithm for 10 million iterations on the original recursive implementation (without tree pruning) and averaged the results over 10 separate trials. In addition, all graphs were produced by taking 1000 equally-spaced samples from the full result dataset. I obtained the following results for 2-player Kuhn Poker:

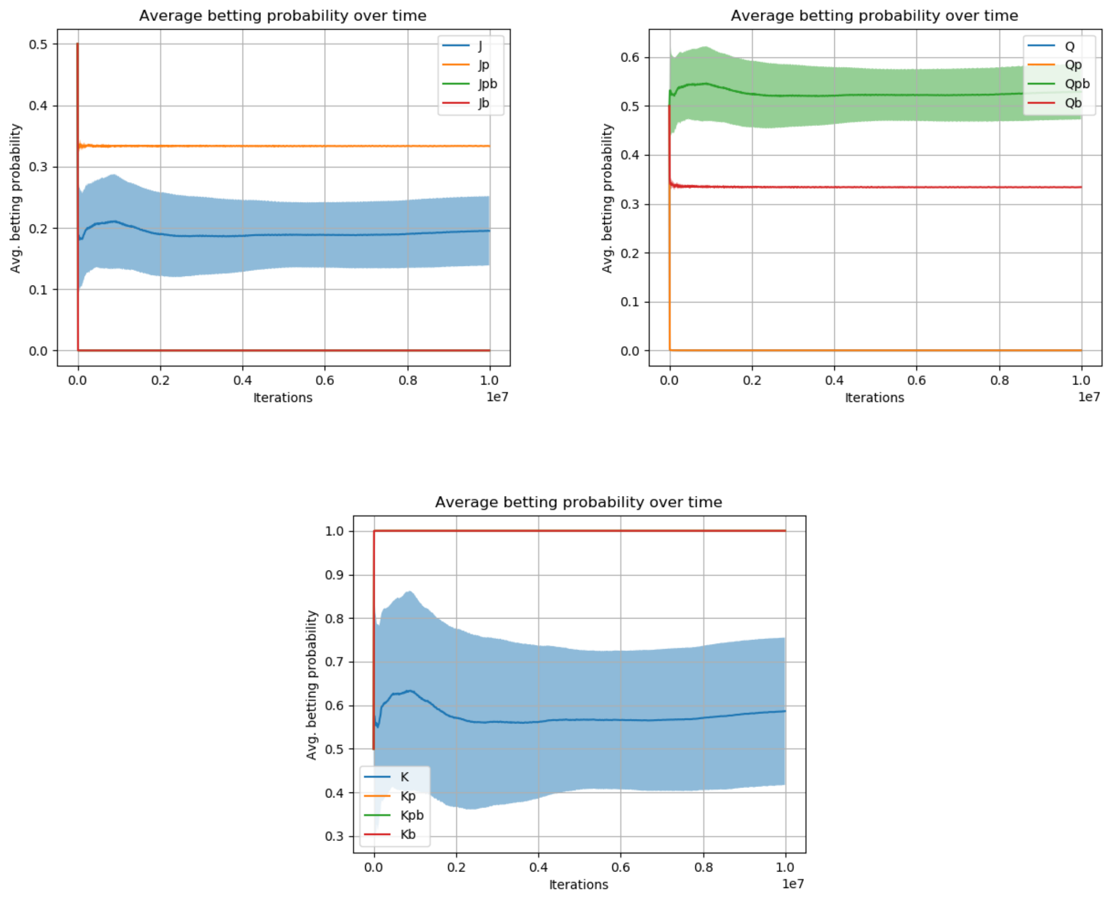


Figure 7.2: Betting probabilities for each decision node with Joker, Queen and King

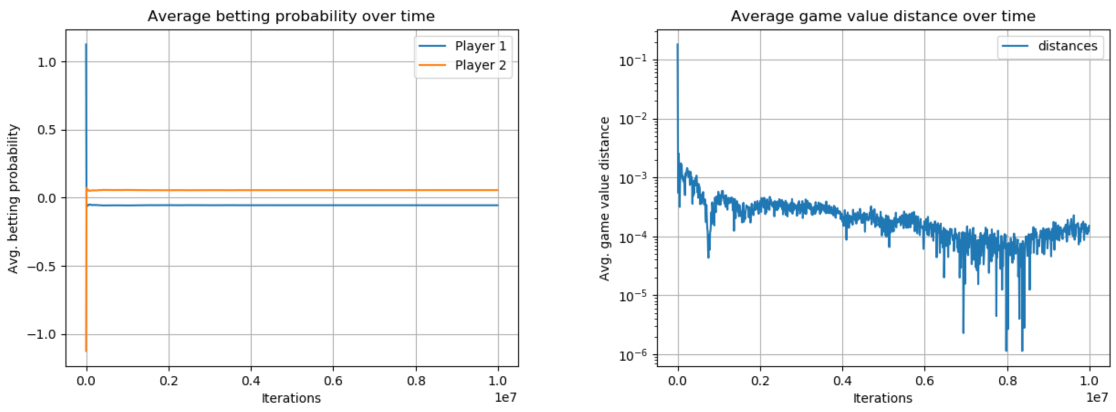


Figure 7.3: Game values for each player and the overall ϵ-Nash values

In the above figures, we can see that the betting probabilities for some decision nodes have relatively large error rates compared to others. This is expected Nash strategies associated with these decision nodes involve free variables, whose values can vary significantly without affecting the Nash equilibrium.

For the 3-player version, it was not practical or particularly useful to graph every single information sets' average strategy. I have instead only plotted the average game values and ϵ-Nash value over each iteration for 3-Player Kuhn Poker:

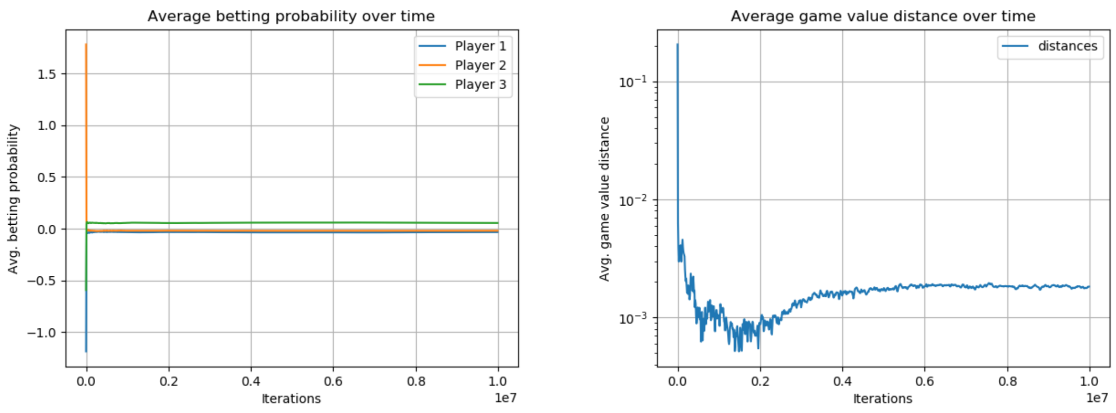


Figure 7.4: Game values for each player and the overall ϵ-Nash values

In both 2 and 3-player Kuhn Poker, it can be seen from their respective graphs that the average game values for each player is deduced to a high degree of accuracy relatively quickly. With more iterations, the ϵ-Nash equilibrium will gradually improve before stabilizing.

7.2.1 Optimisations

I first wanted to investigate the effects of using tree pruning in trying to find an ϵ-Nash equilibrium more quickly. As in previous experiments, I ran the CFRM algorithm for 10 million iterations averaged over 10 trials using the recursive implementation for both Kuhn Poker variants. I repeated the same experiment for the iterative implementation and compared the results of using tree pruning and not.

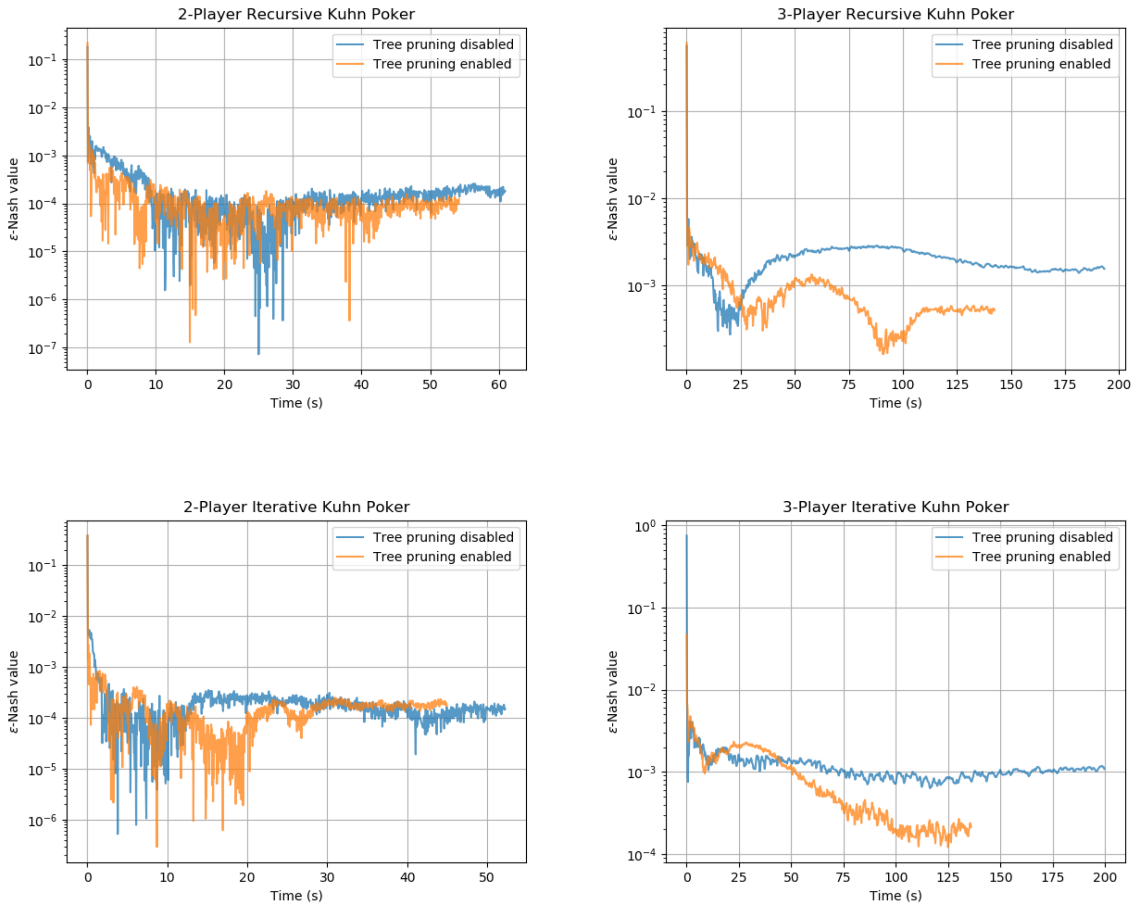


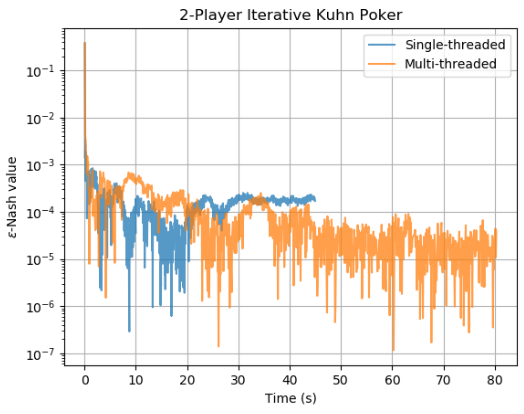
Figure 7.5: Tree pruning comparison for recursive and interative Kuhn Poker

Implementation	Game version	Nodes reached (TP disabled)	Nodes reached (TP enabled)	Percentage reduction
Recursive	2-player Kuhn Poker	9×10^7	6.76×10^7	-24.9%
	3-player Kuhn Poker	2.5×10^8	1.61×10^8	-35.7%
Iterative	2-player Kuhn Poker	9×10^7	6.70×10^7	-25.5%
	3-player Kuhn Poker	2.5×10^8	1.62×10^8	-35.1%

Table 7.1: Tree Pruning results

As we can see in Figure 7.5 and Table 7.1, tree pruning provides a significant amount of speedup for 2 and 3-player Kuhn Poker for both recursive and iterative implementations. The speedup is likely greater in the 3-player version as it's game tree is larger and thus provides more opportunities for the algorithm to prune certain branches. Furthermore (especially in the case of the 3-player version), the ϵ-Nash value obtained after many iterations with tree pruning is often far smaller than without, as can be seen in Figure 7.5. I hypothesise that this is due to having the algorithm only focus the branches that actually contribute to the final game values while ignoring branches that do not.

Another optimisation I wished to investigate was whether the implementation of multithreading led to any significant performance improvements. To do this, I ran a similar experiment as before in which I ran the iterative CFRM implementation for 10 million iterations over 10 trials with multithreading enabled and disabled respectively. Tree pruning was kept enabled in both cases.



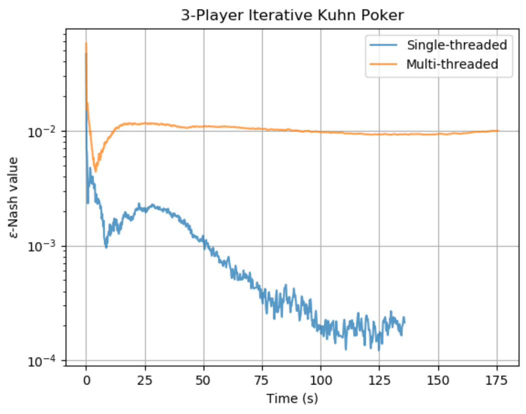


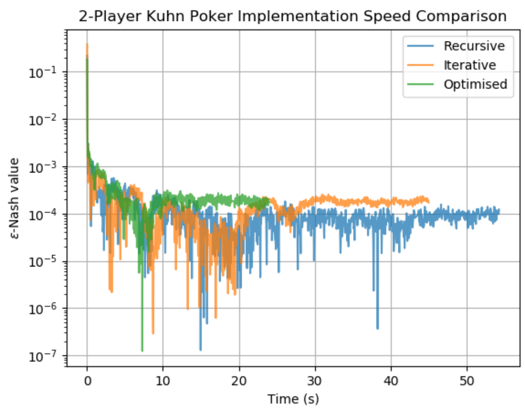
Figure 7.6: Implementation comparison for multithreaded and non-multithreaded CFRM

Game version	Avg. calculation time (s) (multithreading disabled)	Avg. calculation time (s) (multithreading enabled)	Percentage speedup
2-player Kuhn Poker	44.9	80.3	-44.0%
3-player Kuhn Poker	135.9	175.7	-22.7%

Table 7.2: Multithreading results

In this case it can be seen from Figure 7.6 and Table 7.2 that multithreading in both game versions leads to a significant decrease in performance. The problem is more acute for the smaller 2-player game in which performance drops by 44.0%, compared to 3-player where it only drops by 22.7%. From analysis done while the algorithm was running, it seems that the overhead in constantly assigning threads to new jobs outweighs the benefits of parallelization of the algorithm. I believe that the performance drop-off was less severe in the 3-player version as the average number of nodes on each level is greater than in 2-player, so a greater quantity of work could be parallelized.

Following from the failure of attempting to use multithreading to achieve performance gains, I implemented a third version (based on the iterative algorithm) in which all the optimisations I discovered from performance profiling were implemented. As before, I ran the algorithm on this new version for the same number of iterations and trials, then compared it's performance to the old recursive and iterative versions. In each case, tree pruning was enabled and multithreading for the iterative and optimised versions was disabled.



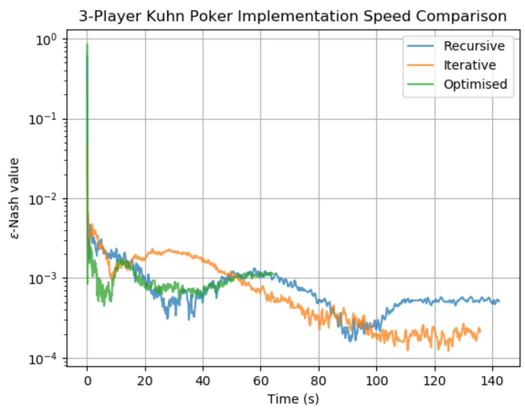


Figure 7.7: Implementation comparison for recursive, iterative and optimised for Kuhn Poker

Implementation	2-player Kuhn Poker		3-player Kuhn Poker	
	Avg. calculation time (s)	Percentage speedup	Avg. calculation time (s)	Percentage speedup
Recursive (TP disabled)	60.9	—	193.2	—
Recursive	54.2	+12.4%	138.6	+39.4
Iterative	44.9	+35.6%	135.9	+42.2%
Optimised	24.0	+154.1%	64.2	+201.0%

Table 7.3: 2-player Kuhn Poker speed comparison results

Figure 7.7 and Table 7.3 both demonstrate that the additional optimisations discovered have vastly improved performance. Compared to the original recursive implementation, the optimised version sees threefold improvement in performance. Compared to the original iterative version the optimised algorithm was based off, the 3-player version once again sees greater relative performance gains than the 2-player version (111.7% vs 87.5% speedups for each version respectively).

8 Conclusion

8.1 Project Achievements

The principal achievement of this project was the successful implementation of the Regret Minimization and CFRM algorithms to Rock-Paper-Scissors and both Kuhn Poker variants respectively. Using these algorithms, I was able to empirically determine an ϵ-Nash strategy for each player where $\epsilon < 0.01$. This work can be extended to solve more complicated games through reinforcement learning where a mathematical solution proves too difficult to discover.

Another major achievement was in successfully discovering methods in which the CFRM algorithm could be sped up. Application of all these optimizations enabled a 3x speedup compared with the original recursive implementation. Even though some approaches such as multithreading did not ultimately work, they still provided valuable insight when investigating other optimizations.

8.2 Future Work

Due to the time constraints, I was not able to pursue every avenue in finding ways of optimizing the CFRM for hidden-information games. There are several other extensions to this project that can be addressed in the future:

- **Improve tree pruning**: While I was able to succesfully implement tree pruning for iterative CFRM, it did not produce the same amount benefit as in recursive CFRM as I was unable to exclude unnecessary work from being done. I believe with further investigation, the problems with iterative tree pruning can be rectified.
- **Monte Carlo CFRM**: MCCFRM is a variant of the CFRM algorithm which samples a

portion of the game's information sets on every iteration instead of going through the entire game tree. While I had investigated usage of this algorithm, I lacked the time to implement this. I have an interest in discovering if this algorithm can be used to provide any potential speedup.

- **Investigate more complex games**: So far I have only implemented CFRM with 2 and 3-player Kuhn Poker. I am interested in learning how well the benefits of various optimisations scale for more complicated games with larger game trees. I am also interested in learning if multithreading provides speedup in such games when it does not in simple ones.

- **GPU implementation**: GPUs are known to be much faster when processing large volumes of data in parallel compared to CPUs, and are thus of great interest in machine learning. In the future, I would like to further extend a multithreaded CFRM implementation to utilize GPUs instead, to investigate if we could achieve even further speedups in processing time.

Bibliography

Duane Szafron Richard Gibson, Nathan Sturtevant. *A Parameterized Family of Equilibrium Profiles for Three-Player Kuhn Poker.* 2013. URL: http://poker.cs.ualberta.ca/publications/AAMAS13-3pkuhn.pdf.

Kuhn, H. W. and A. W. Tucker. *Contributions to the Theory of Games. Simplified Two-Person Poker.* Vol. 1. Princeton University Press, 1950. ISBN: 978-0-691-07934-9.

Kullback, S. *Information Theory and Statistics.* Dover Publications, 1959. ISBN: 0-8446-5625-9.

Martin Zinkevi Michael Johanson, Michael Bowling and Carmelo Piccione. *Regret Minimization in Games with Incomplete Information.* URL: http://poker.cs.ualberta.ca/publications/NIPS07-cfr.pdf.

Neller, Todd W. and Marc Lanctot. *An Introduction to Counterfactual Regret Minimization.* 2013. URL: http://modelai.gettysburg.edu/2013/cfr/cfr.pdf.

Reis, João. *A GPU implementation of Counterfactual Regret Minimization.* 2015. URL: https://sigarra.up.pt/feup/en/pub_geral.show_file?pi_gdoc_id=418762.

Skymind. *A Beginner's Guide to Deep Reinforcement Learning.* URL: https://deeplearning4j.org/deepreinforcementlearning#define.