Learning to Play Leduc Hold'em Using Reinforcement Learning

UNIVERSITY *of* LIMERICK

OLLSCOIL LUIMNIGH

Department of CSIS

Bachelor of Science in Computer Systems

Author: Jamie Mac Manus

ID: 15147312

Supervisor: J.J Collins

Abstract

This report investigates the possibility of applying reinforcement learning techniques to imperfect information games such as poker. A detailed research process is documented which begins with high-level reinforcement learning concepts and leads to specific algorithms that can be utilised to solve games like poker. After review of the relevant literature, a modified version of Monte Carlo Tree Search was selected as the algorithm that would be utilised to solve our poker variant, Leduc Hold'em. The project set out to re-create the results of a recent Ph.D. thesis by Johannes Heinrich that demonstrated the effectiveness of this method. A number of experiments were conducted, utilising various metrics to measure the performance of the poker agent created. The results of these experiments were then recorded and analysed. A prototype game was created to allow users to interact with an instance of the trained agent. Monte Carlo Tree search was found to be a relatively effective method for strategy development in Leduc Hold'em agents. The agent performing well against human opponents and achieving an exploitability level of 2.2 in our final experiment.

Acknowledgements

First I would like to thank my FYP supervisor, Mr. J.J Collins for his constant support, feedback, and guidance on the project. I came to him with the broad aim of applying reinforcement learning to poker and through his expertise and engagement, he was able to help me find a path for the project that treads the line of ambition and feasibility. Throughout the year J.J and I organised weekly meetings in which he provided invaluable feedback that led the project forward. I also greatly appreciate his patience as I worked to gain familiarity with a complex new area of computer science.

I would also like to thank my family for their support, and to my friends with whom many fun times were had throughout my time in college.

I would like to thank Rory Egan, Dan Noonan and Kevin Moynihan with whom I collaborated on a number of projects during the four-year course. Their effort, talent, and insights made project work a painless task throughout the years.

I would like to thank Dr. Jim Buckley for his work as FYP coordinator. Finally, I would like to thank the CSIS faculty and staff who have all contributed to a very positive four years of learning in this department. This is especially the case for all the lectures and TAs that I have had over the years who helped me develop a foundational CS knowledge that stood me in good stead for this project.

Contents

List of Figures

List of Abbreviations

- **ACPC** - Anual Computer Poker Competition
- **BR** - Best Response
- **DP** - Dynamic Programming
- **FP** - Fictitious Play
- **FSP** - Fictitious Self Play
- **GWFSP** - Generalised Weakened Fictitious Self Play
- **MC** - Monte Carlo
- **MCTS** - Monte Carlo Tree Search
- **MDP** - Markov Decision Process
- **ML** - Machine Learning
- **NFSP** - Neural Fictitious Self-Play
- **POMDP** - Partially Observable Markov Decision Process
- **RL** - Reinfocement learning
- **TD** - Temporal Difference
- **UCB** - Upper Confidence Bounds
- **UCT** - Upper Confidence Bounds Applied to Trees

Chapter 1

Introduction

1.1 Overview

Since the inception of machine learning(ML), games have been a key problem area that has seen a lot of focus in academia. For decades games have been used as a platform to test and develop algorithms that have gone on to provide invaluable services that are used in peoples everyday lives. For example(Tesauro 1995) demonstrated the value of the reinforcement learning techniques with his backgammon application. Today we see these techniques starting to permeate complex industry domains such as vehicle automation(Desjardins & Chaib-Draa 2011). The ability to contribute to this great history was a large motivator when it came to choosing this project.

The game that was initially selected for this project was Texas Hold'em which is a popular variant of poker. In chapters 1 and 2 we will primarily refer to Texas hold'em due to the fact that the majority of the literature is concerned with solving this version of poker. In chapters 3 and 4 we implement and analyse a solution for a simplified version of Texas hold'em named Leduc hold'em. As such we primarily refer to Leduc Hold'em in these chapters.

Although this report will discuss ML and how it applies to games in general, the primary focus will be on ML techniques when applied to Texas

hold'em, or variations of it. For the past number of years, a method called Counterfactual Regret Minimization (CFR) has dominated the top spots in the Annual Computer Poker Competition(ACPC). For example, the 2018 heads-up champion, named Slumbot utilised these methods(Jackson 2013). CFR is an algorithm that allows computation of a strategy through self-play(Zinkevich et al. 2008). When applied to large imperfect information games such as Texas hold'em CFR generally relies on the creation of abstractions of the game(Neller & Lanctot 2013). These abstractions are used to reduce the size of the game's state space. CFR is then applied to these abstractions in order to learn a strategy. It is thus critically important that these abstractions accurately represent the full game's mechanics. As such a high degree of knowledge of the game is required in order to create an accurate abstraction.

There have also been attempts to tackle Texas hold'em using a ML paradigm called reinforcement learning (RL). The great appeal of these approaches is that they do not require us to generate custom abstractions of the game. As a result of this, it was decided that this project would utilise RL techniques to tackle the problem.

RL is a way of programming agents by reward and punishment without needing to specify how the task is to be achieved(Kaelbling et al. 1996). On a surface level, RL seems like a perfectly reasonable method for solving games such as Texas hold'em. However, Texas hold'em is an imperfect information game. This means that we do not know the entirety of the state information at any given time i.e we do not know the values of the opponent's cards. Thus, from a RL perspective, we do not know the actual state from which we are choosing actions. This makes pure reinforcement learning strategies impractical for these types of games.

However, there has been some success when custom reinforcement learning methods have been implemented. In one case linear programming techniques and RL were used to tackle a simplified version of the game(Dahl 2001). In another case, RL was combined with supervised learning and game

theoretic techniques(Heinrich & Silver 2016) to develop an algorithm called Neural Fictitious Self-Play(NFSP). This method became the focus of the project for a time before a decision was made that the broad range of techniques involved would add too much risk to the project. When it was discovered that the same researcher, Johannes Heinrich had applied an RL related search algorithm called Monte Carlo Tree Search (MCTS) to several variations of poker(Heinrich & Silver 2015) this algorithm became the primary focus of the project. This algorithm was of great interest for a number of reasons. Firstly, this relatively new technique had been the basis for the tremendous improvement in the performance of computer Go programs over the last decade. The culmination of which was the AlphaGo project that defeated the highest ranked human Go player in the world(Silver et al. 2016). Secondly, this approach had shown the best results of any of Heinrich's documented approaches with the developed agent placing second in the 2014 ACPC . Finally, the lead researcher of the AlphaGo project, David Silver had collaborated on Heinrich's work thus adding significant weight to the research.

Thus in this project the MCTS algorithm will be applied the chosen test domain, the game of Leduc Hold'em. This implementation will be the basis for a number of experiments and a prototype game.

1.2 Objectives

1.2.1 Primary Objectives

Although this project will be largely research-based, the initial goal is to create a Texas hold'em playing agent. Due to the fact that Texas hold'em has an extremely large state space, combined with the fact that it is an imperfect information game, the initial goal will be to tackle a simplified version of the game. Specifically, Leduc hold'em will be used for this project. This version of hold'em consists usually of a six card deck and only one private card, compared to two in Texas hold'em.

In(Heinrich & Silver 2015) a metric called exploitability was used. This is a measure of how the agent's strategy fares against the best responses to that strategy. In other words, if the opponent knows our strategy, and can take the best possible action in every state in order to maximise their potential gain in reward, exploitability is the average amount that they would gain from doing so. For Leduc Hold'em, with a 6 card deck and 500,000,000 iterations, the initial exploitability was 2 and converged to .0223. As such the success criteria for our initial iteration of the game is to replicate these results, with an allowance for hardware differences that may impact computational speed.

It is also my goal to create a product that will be fun and useful for the general public. As such another objective will be to create an application that will allow users to play heads-up against the final product.

1.2.2 Secondary Objectives

As this project is very specific and academic, one of the larger challenges will be to gain a strong knowledge of the domain. This means learning the history of RL, the types of problems that it has been used to solve and the specific details of different RL algorithms.

A successful project will require a high degree of knowledge from the broader domain of RL. However, it is also the case that I must become closely familiar with the existing academic literature in the area of RL with respect to imperfect information games. This will allow me to avoid taking approaches that have previously shown to fail as well as allow me to add value to the existing literature whether that be through literature review or through my own experimental findings.

1.3 Methodology

The methodology for this project can be summarised as follows:

1. **Identify Research Objectives**: The first step of this project was to

define research goals. Although it was established that an agent for a variation of poker would be created early in the project,

2. we did not yet know what variation we would be tackling nor the method that would be applied to do so. Identifying exactly what would be done and how was the first big step in this project.

3. **Approximation of Systematic Literature Review**: As a result of the scope of the project a

4. full-scale systematic literature review was not required. However, it was still important to gain a strong understanding of the relevant literature. This was achieved by utilising various search terms and in Google Scholar, saving any relevant papers and reviewing their contents. When particularly interesting papers were found many of their references were then followed and read to give a deeper understanding.

5. **Identify Technique for Implementation**: When defining our research objectives the broad

6. area of reinforcement learning had been selected but

7. **Design and Implement Prototype**: When the pertinent research objectives had been defined and the method through which they would be achieved had been found, the next step was to design and implement the prototype. This involved closely following the algorithm defined in the chosen research paper and converting that to functional code.

8. **Empirical Studies**: When the prototype had been developed the next step was to begin to measure its performance in order to estimate if defined research objectives had been met.

1.4 Overview of the Report

This report is subdivided into a number of chapters each with a distinct purpose. The introduction gives an overview of what the project is about, the goals of the project, and how these goals will be achieved. The background will present the knowledge that is required to proceed to the later stages of the report with sufficient understanding. It will also have the purpose of documenting the research that was done throughout the course of the project. The implementation chapter will discuss how we apply the knowledge that was acquired in the background to our agent implementation. This chapter gives a deep dive on the code that was implemented and explains how this code operates and relates to the abstract algorithms discussed in chapter 2. The empirical studies chapter then discusses the empirical results gleaned from the work in chapter 3. The metrics that were used to evaluate our poker agent are discussed, the parameters used for the algorithm in each experiment are outlined and analysis of the results is given. Finally, the conclusions chapter gives a retrospective look at the project. With an overview of how the project met its goals, and a discussion of how the project could be expanded in the future to add further value.

1.5 Contribution

The research done throughout this report should contribute to the academic area of reinforcement learning applied to poker in various ways. First, we consolidate summaries of a number of approaches that can be taken to solve this problem into a single resource. Second, a more in-depth description of Heinrich's 2017 application of MCTS to Leduc Hold'em is provided through coding snippets and detailed explanation. A great deal of research was required for this implementation with multiple sources being called upon to piece together our implementation, thus having our code open source and available to the world may aid future researchers in the same area.

1.6 Motivation

For the last several years, I have played poker recreationally with friends or online. It became more of an interest of mine as I started to explore the mathematical basis for the game and how players could use their knowledge, intelligence, and temperament to gain an advantage in a game that, on the face of it, seemed to be largely based on chance. I spent some of my free time researching different aspects of the game. This included gaining some basic knowledge like the probability of making drawing hands as well as learning more technical aspects of the game such as how to calculate the expected value of hands, or how to narrow down one's opponent's range of possible hands.

Concurrent with the development of this interest I was also becoming more and more interested in the area of ML . ML and the development of artificial intelligence is possibly the most glamourised area of computer science. However, this is probably for good reason because there is something intrinsically interesting about machines that can learn to solve a problem on their own, without direct instructions from a human. The fact that ML has made so many strides in recent years was another cause of interest in this area of computer science, especially as the practical viability of ML as a means of tackling a wide array of problems in industry continues to increase.

As a result, the merging of these two interests as the basis for my final year project seemed like an obvious choice.

Chapter 2

Background

The aim of this chapter is to give the reader background information on certain areas of machine learning and game theory in order for them to understand the rest of the report. There will also be an in-depth discussion of existing literature that relates to machine learning in texas hold'em agents.

Machine learning is an area of computer science that tackles the construction of computer programs that improve with experience(Mitchell et al. 1997). The term was coined by Arthur Samuel in a paper in which he discussed machine learning methods using checkers(Samuel 1959). Since then there has been a great deal of advancement in the field. Some of the notable early contributions being the discovery of recurrent neural networks in 1982, the advancement of reinforcement learning by the introduction of Q-Learning in 1989 and the development of a backgammon-playing agent using neural networks and temporal difference learning(Tesauro 1995). Recently some of this early academic work has culminated in more practical achievements such as Facebook's DeepFace system which, in 2014, was shown to be able to recognise faces at a rate of 97.35% accuracy, a rate that is comparable to that of humans. Another example of recent achievement is Google's AlphaGo program which, in 2016, became the first program to beat a professional human player.

It should be becoming clear that machine learning can be a solution to

a wide array of problems and as both hardware and software continue to improve its reach will only continue to grow. Machine learning systems are starting to become a key component of many companies business model. Since certain machine learning techniques are great at prediction, machine learning has been widely used for content discovery by companies such as Google and Pinterest. Other business applications include the use of chatbots as a part of customer service, self-driving cars and even in the field of medical diagnostics.

2.1 Reinforcement Learning

The early research for this project yielded reinforcement learning as the most suitable machine learning paradigm for the problem of Texas hold'em. However, in order to understand both reinforcement learning and how it would apply to the chosen problem, in-depth research was required. This research included a Udemy course(LazyProgrammerInc. 2018) as well as reading in part the book Reinforcement Learning: An Introduction(Sutton et al. 1998).

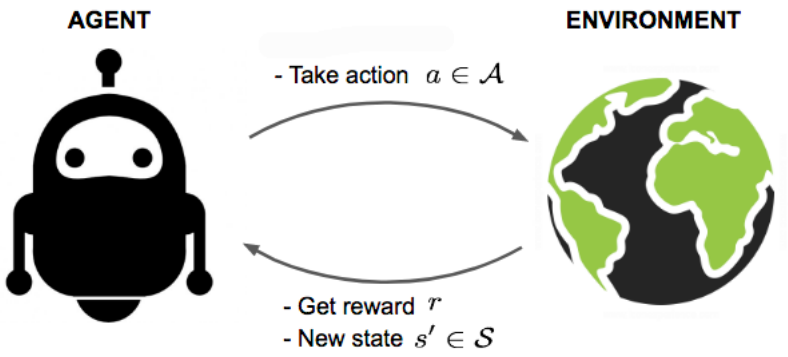

Figure 2.1: Reinforcement Learning - adapted from(Weng 2018*a*)

As mentioned in the introduction, reinforcement learning is a method of programming agents by reward and punishment without needing to specify how the task is to be achieved. As such the primary components of a reinforcement learning problem are an agent which exists in an environment.

From a simplified perspective, we can think of the environment as a set of states, actions, and rewards. The interaction of the agent and the environment using these three components is shown in figure 2.1. The objective for the agent is to maximise cumulative reward. This is done by developing a policy that will dictate which actions should be taken in each state.

2.1.1 Explore-Exploit Dilemma

When it comes to reinforcement learning one of the first questions that should be asked is how the state space should be explored. An example that is often used to conceptualize this problem is the multi-armed bandit problem. Let's say an agent is in a room with a number of gambling machines. Each of these machines has an arm that, when pulled will return a reward of 0 or 1 based on some underlying probability(Kaelbling et al. 1996) (see figure 2.2). The agent has a limited number of total pulls. So the question becomes how should these pulls be distributed in order to maximise return? Well, first, enough exploration must be performed such that the machine with the best reward probability is found and second, this machine must be maximally exploited.

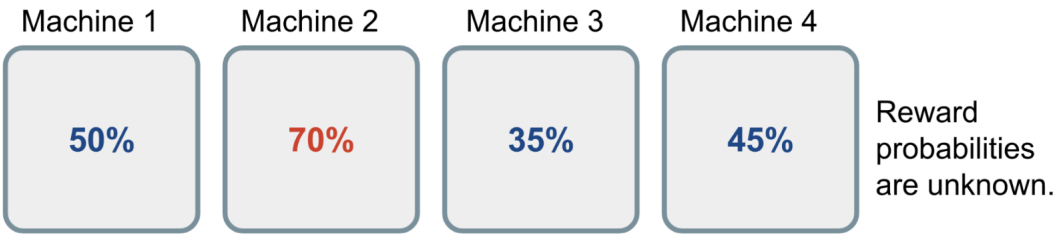

Figure 2.2: Multi Armed Bandit - adapted from(Weng 2018*b*)

There are a number of approaches that can be taken to solve this problem, two of these methods will now be briefly discussed.

ϵ-Greedy Solutions

The first approach is the ϵ-greedy strategy. This approach was first proposed in(Watkins 1989) and is a very simple and widely used method. The ϵ-greedy strategy involves choosing a random lever some proportion ϵ of the time, and choosing the lever that has been established to give the highest reward for the rest of the time.

There are a number of variations of this method, the first being the ϵ-first strategy. With this strategy all random choices are taken first, thus evaluating the best bandit, after which this bandit is exploited. However, as stated in(Vermorel & Mohri 2005) this simple approach is sub-optimal because asymptotically, the constant factor prevents the strategy from getting arbitrarily close to the optimal lever.

This is where the ϵ-decreasing strategy becomes useful. Here, the proportion of random lever pulls decreases with time. Generally, if the initial epsilon value is ϵ_0 then the epsilon value at time t will be $\epsilon_t = \frac{\epsilon_0}{t}$.

Upper Confidence Bounds

Another approach that can be used is called the upper confidence bound(UCB) method. With this method, an initial optimistic estimate of the reward of each bandit is assigned. After which a greedy approach is taken. Less explored bandits will have an artificially higher reward estimate and thus they will be greedily chosen, thus allowing us to evaluate each of the bandits.

In the context of reinforcement learning, state space exploration through the ϵ-greedy approach is generally sufficient.

2.1.2 Markov Decision Processes

In the last section, some methods that can be used to explore environments have been established. The next step is to discuss in more detail how reinforcement learning environments, and their interaction with reinforcement learning agents, are modeled. Generally finite Markov decision processes (fi-

nite MDPs) are used. Markov decision processes provide a formal mathematical framework for sequential decision making, where actions influence immediate rewards as well as subsequent situations(Sutton et al. 1998). MDPs allow us to create an idealized model of reinforcement learning problems and thus precise theoretical statements can be made.

MDP Dynamics

As mentioned earlier, reinforcement learning problems consist of an agent and an environment interacting. Markov decision processes can be looked at in a similar way. However, there are a number of additional factors that must be considered in order to paint a complete picture.

The problem can be thought of as consisting of a set of discrete time steps. At each time step, the environment supplies the agent some information about the state, S_t. Using this information the agent chooses an action, A_t. Then, as a result of the action, the environment will supply the agent with a reward, R_t, as well as a new state. As such the process of interaction between the agent and the environment can be seen as a trajectory of states, actions and rewards(Sutton et al. 1998):

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2 \dots \tag{2.1}$$

Thus far it has been stated that states, actions, and rewards are related, however questions still exist as to the exact workings of this relationship. The answer is that finite MDPs contain a discrete probability distribution that determines the likelihood that we will reach the state s' and receive reward r at time t based on the previous action a and state s:

$$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.2}$$

In simplified terms, this means that for a given state-action pair (s, a), the probability of us reaching some new state s' and receiving reward r is determined by the MDPs probability function Pr.

This four argument probability function completely characterizes the dynamics of the MDP and from it any other information about the environment can be obtained(Sutton et al. 1998).

MDPs and Learning

The goal of the agent in an MDP is to learn how to maximise the cumulative reward received when traversing the environment. In some cases the MDP will be traversed until a terminal state, T, is reached. This type of MDP reflects episodic tasks that will always terminate. In this case cumulative reward,G_t can be calculated as follows:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.3}$$

However, in other cases continuous tasks are modelled. The problem here is that, if the same method of calculating G_t is used then in this case G_t will always eventually sum to infinity, regardless of whether good or bad actions are being selected. As such a concept called discounting must be introduced. With this approach, the aim is to maximise the sum of future discounted rewards. Thus γ, a parameter with a value between 0 and 1, is utilised. As such, for continuous tasks modeled as MDPs our cumulative reward is as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \tag{2.4}$$

Based on our specified value of γ the weighting of future rewards can be adjusted. For example, if a low value for γ (eg .5) is used then the value of rewards more than a few time steps in the future will be very low.

Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) is used to model environments that are not fully observable(Kaelbling et al. 1996). POMDPs extend MDPs by including a set of observations, O. There is also an observation function, $P(O_t = o|S_t = s)$, that determines the probability of making a

certain observation in a certain state. Hold'em can be modeled as a POMDP due to the fact that is an imperfect information game.

However, it is possible that an agent in a POMDP environment can remember the sequence of observations and actions that lead to the current state. This is a sufficient statistic of its experience and can thus define a complete information state(Heinrich 2017). As such a POMDP can be reduced to it's underlying MDP by using these full history information states and also extending the relevant transition and reward functions.

2.1.3 Policy Evaluation and Policy Improvement

As mentioned above the primary focus of reinforcement learning is to find a policy (denoted by π) that allows the agent to take actions in states that lead to the maximum possible reward. There are two primary problems that must be solved in order to do so.

The first is called the prediction problem, or policy evaluation. This involves computing the values of states given some arbitrary policy(Sutton et al. 1998). For example, a state would have a high value if the reward for reaching that state was high. A state would also have a high value if the current state was only one action away, according to the supplied policy, from a state that renders a high reward. However, a state would have a low value if, according to the policy, there was no path to a state that would return a positive reward in the foreseeable future.

The second problem is known as the control problem or policy improvement. This involves changing the policy in order to improve our cumulative reward. The policy improvement process can only occur when policy evaluation has been performed. Let's say, after our evaluation step, the value of some state s is known. Note that this value is calculated with the condition that some action a is taken in state s. But, if some other action a' is taken, would this render a higher value for s? If the answer is yes then the policy is updated.

These two operations can be seen as the core of reinforcement learning.

In the next section, different reinforcement learning methodologies will be discussed. Some of the main differences are in how method each approaches the prediction and control problems.

2.1.4 Dynamic Programming

Dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process(Sutton et al. 1998). DP is not widely used in practical reinforcement learning applications due to its assumption of a perfect MDP and it's high computational requirements. Despite this, it is very important from a theoretical standpoint as it serves as an introduction to a number of important reinforcement learning concepts. Furthermore, it provides a basis for many algorithms that are used in practical reinforcement learning applications.

Policy Evaluation in Dynamic Programming

When discussing policy evaluation a state-value function or a value function is referred to. This is simply the mapping of states to their corresponding values and is denoted by v.

Since the environments' dynamics are completely known, an iterative solution to finding the value function can be applied. If a series of approximate value functions $v_0, v_1, v_2, ..$ are considered. The initial value function, v_0 is chosen arbitrarily and each successive generation is obtained by using the Bellman equation for v_π as an update rule(Sutton et al. 1998):

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \tag{2.5}$$

In order to produce each successive approximation of v_{k+1} from v_k the operation outlined to each state s is applied. As shown above the new value for s is based on a combination of the expected immediate rewards(R_{t+1}), and the expected values of each of the states that can be transitioned to(S_{t+1})

given policy π. It can be shown that as $k \to \infty$ v_k will converge to v_π, the correct value function for policy π. This algorithm is called *iterative policy evaluation*(Sutton et al. 1998).

Policy Improvement in Dynamic Programming

Since the value of following v_π is known, this information can be used to determine how this policy should be modified in order to improve its value. Assume that π is a deterministic policy. This means that $\pi(s)$ will return some action that must be taken. Now the question becomes what if some other action $a \neq \pi(s)$ is taken? This depends on whether or not choosing this action, and then continuing to use the existing policy will improve the value of the policy. If it does then this new action will be chosen.

The logical extension of this approach is to apply it to each state and each possible action. As a result, what appears to be the best action at each state will be selected. The new greedy policy π' can be denoted as:

$$\pi'(s) = argmax\, \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \tag{2.6}$$

Essentially here the value of each available action in the current state is determined using the same operation as outlined in the policy evaluation phase. Then the argmax function will select the action with the highest value. Finally, this action is assigned to be the one that will be chosen in state s, according to the new policy π'.

Note that in this section an algorithm has been outlined with respect to deterministic policies, however, a lot of times in reinforcement learning deals with stochastic policies. This means that actions are taken in states according to some probability distribution rather than always choosing the same action in a particular state. This is not a problem as the ideas mentioned apply equally well to stochastic policies.

2.1.5 Monte Carlo

Monte Carlo(MC) methods are a wide range of algorithms that rely on random sampling in order to obtain results. Such a method could be applied in the case of having an array of 1 trillion elements that could be either 1 or 0. In order to calculate the number of 1s in such an array iteration could be used. However, this would be extremely computationally costly. Instead, random indices could be generated to sample the array. A count of both the number of 1s found and the number of indices generated would be maintained. Based on these figures it would be possible to estimate the total number of 1s in the array. Over time as more samples are taken the estimate would converge towards the true value.

Monte Carlo based reinforcement learning techniques apply this method in the policy evaluation step. In Monte Carlo, unlike dynamic programming, there is no assumption of complete knowledge of the environment. Monte Carlo methods require only experience. Sequences of states, actions, and rewards are sampled from interaction with the environment(Sutton et al. 1998). These sequences are called episodes. Monte carlo evaluation is an episodic process. This means that action values are only updated after an episode has completed.

Monte Carlo Policy Evaluation

In Monte Carlo methods a fundamentally different approach to policy evaluation is taken. As mentioned this method is focused on using episodic experience. In order to evaluate a state, the rewards returned after visiting that state are averaged. As more returns are observed the average will converge to the actual expected value of the state.

It is worth noting that a state s could be visited more than once in an episode. As such, the returns can either be averaged following the first visit to s or averaged after each visit to s. These two methods are called *first-visit* and *every-visit* respectively.

Monte Carlo Action Values

In Monte Carlo methods, the lack of a model means that state values are insufficient in order to obtain a policy. Rather state-action pairs are generally evaluated. This mapping of state-action pairs to values is referred to as the q function or the action value function. In this case, the evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π(Sutton et al. 1998).

The method for policy evaluation using state-action pairs is almost identical to that outlined above. The only difference being that instead of averaging rewards for each state, rewards for each action taken when a state is visited are averaged. There is one problem with this approach in the context of deterministic policies. The problem being that in following a deterministic policy rewards will only be returned for a single action, thus only one action value estimate will be improved. In order to negate this problem, it can be specified that every episode begin in a state-action pair, with the probability of starting in each state-action pair being non-zero. This is called the *exploring-starts* method. Another approach would be to ensure that a stochastic policy is used with the probability of selecting each action being non-zero.

Monte Carlo Policy Improvement

In Monte Carlo methods, the overall policy improvement algorithm is the same as outlined in the dynamic programming section. That is, it alternates between modifying the value function to more closely approximate the current policy, and using the value function to improve the policy. However, as mentioned, in the case of Monte Carlo a Q function is used as shown in figure 2.3.

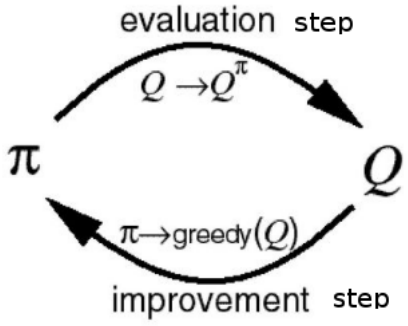

Figure 2.3: Monte Carlo Policy Improvement - adapted from(Lee 2005)

2.1.6 Temporal Difference Learning

The final reinforcement learning method that will be discussed is the Temporal Difference(TD) learning method. TD learning combines dynamic programming(DP) and and Monte Carlo(MC) ideas(Sutton et al. 1998). Like MC, learning occurs directly from experience, without a model of the environment's dynamics. However, like DP state value estimates are updated based on other learned estimates, without needing to wait for an episode to complete and the return of some final outcome.

The selective use of different aspects of these reinforcement learning methodologies by TD learning has a number of advantages. The fact that a model of the environment is not needed makes it easier to implement TD methods compared to DP . TD methods are also conducive to solving problems with long episodes or even continuous tasks with no episodes at all due to the fully online nature of this learning algorithm.

TD Learning Policy Evaluation

Unlike MC, with TD learning the value function can be updated at every time step. This is exemplified by the *TD(0)* or *one-step TD* method in which the update is made immediately on transition from state S_t to state S_{t+1}. The more general case of this method is the *TD(λ)* or *n-step TD*. With *TD(0)*

the update rule is as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{2.7}$$

As such the new value for some state S_t is dependant on the previous value of the state($V(S_t)$), along with the reward(R_{t+1}) gained from transitioning to state S_{t+1} plus the discounted(γ) estimated value of that state($V(S_{t+1})$). The sum of the latter is multiplied by α which is a small positive fraction that influences the learning rate.

This rule is applied for each state visited in an episode and for each episode.

TD Learning Policy Improvement - SARSA

At this point, it is worth noting that there are two distinct methods of handling policy improvement. The first is on-policy and the second is off-policy. On-policy reinforcement learning is when the policy being evaluated or improved is the same policy that is used to make decisions. In off-policy reinforcement learning the policy being used to generate behavior is not the same as the policy being evaluated or improved.

SARSA is an example of an on-policy algorithm. The policy improvement mechanism is the same here as outlined in the previous sections. With SARSA, like in Monte Carlo the action-value function($q_\pi(s, a)$) is utilised rather than the state-value function. Thus the policy evaluation step is slightly modified as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{2.8}$$

This update utilises the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, which is where the name SARSA originates.

TD Learning Policy Improvement - Q-Learning

Q-Learning is an off-policy control algorithm. This was one of the early breakthroughs in reinforcement learning as it allows the direct approximation

of the optimal action-value function independent of the policy being followed. The update rule is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma maxQ(S_{t+1}, a) - Q(S_t, A_t)]. \quad (2.9)$$

Here our update rule is similar to that of SARSA apart from the fact that the value of the best action available ($maxQ(S_{t+1}, a)$) is used when updating our action value.

2.1.7 Monte Carlo Tree Search

In this section Monte Carlo Tree Search(MCTS), a decision time planning algorithm will be discussed. This iterative algorithm uses tree search combined with value estimations from MC simulations in order to efficiently solve very large MDPs. One application of this algorithm was in DeepMind's AlphaGo(Silver et al. 2016) which became the first computer program to beat a highly ranked Go champion.

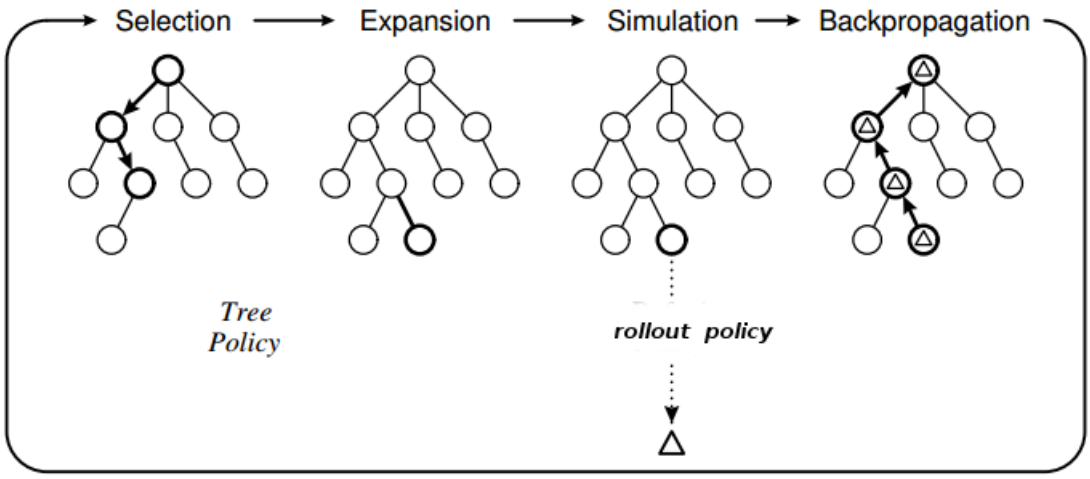

Figure 2.4: MCTS - adapted from(Lim & Yoo 2016)

MCTS is an episodic process. As shown in figure 2.4, there are four key steps involved in each of these episodes. The first step is **selection**. From the root of the tree, the tree policy is used. This policy is based on action

values associated with the edges of the tree, in order to select a child node at each level(Sutton et al. 1998). The second step is **expansion**. When a leaf node is reached in the tree, one or more child nodes may be added based on the actions available at that point. The third step is **simulation**. When the tree has been exited the rollout policy is used in order to provide default behavior outside of the tree and ultimately reach a terminal node from which a reward can be obtained. The final step is **backpropagation**. This step uses the reward obtained to update the values of the tree nodes that were visited during the episode.

In summary, this algorithm uses the simulation step in order to efficiently sample rewards which are then propagated back to nodes of the tree. In later iterations, the tree policy selects which areas of the tree to explore based on these propagated values. In section 2.4 a method for applying MCTS to imperfect information games such as poker will be outlined.

2.2 Game Theory

After taking a deep dive on reinforcement learning and the papers surrounding RL in texas hold'em it became clear that a pure reinforcement learning approach would not be feasible. The main reason for this was the fact that Texas hold'em is an imperfect information game. As outlined by(Dahl 2001):

> Note that the concept of game state values, which is the key to solving perfect information games, does not apply to imperfect information games, because the players do not know from which game states they are choosing.

As such it became apparent that some other techniques would have to be incorporated in order to create a competent texas hold'em agent. After discovering the following two key papers(Heinrich & Silver 2016) and(Heinrich 2017), the initial goal was to utilise the Neural Fictitious Self-Play(NFSP) method as a basis for our agent. Eventually, instead, it was decided that

MCTS would be used as the basis for our agent implementation. NFSP is based heavily on the game theoretic concepts that are outlined in this section, however, the MCTS implementation that is discussed later will rely on a number of these concepts.

2.2.1 Modelling Games

When an attempt is made to solve a game, the way in which the game is modeled must first be understood. It is also important to carefully consider how a form or structure of the model is defined(Myerson 2013). In this section, three important forms that games can take will be discussed. The varying utility of each of the different scenarios will also be outlined.

Extensive Form Games

Extensive form games are a model of players sequential interaction, with explicit representation of a number of key aspects of the game. (Kuhn 1953) formally defined extensive form games as consisting of the following components:

- N - a set of players.
- S - a set of states that represent the nodes of a rooted game tree.
- $A(s)$ - a set of actions for each state representing the edges to the following states.
- U - a set of information states (one set for each player).
- *Player Function* - determines who is to act at a given state.
- *Information Function* - determines which states are indistinguishable for the player by mapping them to the same information state.
- *Return Function* - maps terminal states to each player's return/payoff.

Extensive form games allow us to richly describe game situations. This allows us to uncover characteristic differences between games and the structural features which determine these differences(Kuhn 1953).

Behavioral strategies for players can also be defined. These consist of a probability distribution over actions given information states(Heinrich 2017). This is denoted by $\pi^i(a|u^i)$. If there exists a collection of strategies for all players in the game then this is called a strategy profile, π. π^{-i} is the set of strategies in π not including π^i. This will be discussed later when Heinrich's implementation is discussed in more depth.

Normal Form Games

Normal form games are represented by way of a matrix. Although some information is lost in comparison to extensive form games, normal forms are better suited to the derivation of generalized theorems(Kuhn 1953) and thus have their own utility.

An extensive-form game can induce an equivalent normal form of the same game. This can be done through the generation of a set of deterministic behavioral strategies for each player called pure strategies. Each pure strategy is a full game plan that will determine an action for each situation the player may encounter. Mixed strategies can also be created which define a probability distribution over the players pure strategies. A mixed strategy for player i is denoted as Π^i. When the extensive-form return function is restricted to normal-form the yield is an expected return function. The expected return function for some mixed strategy profile Π is $R^i(\Pi)$(Heinrich 2017).

Sequence Form Games

In order to compute a Nash equilibrium for an extensive form game the game can be converted to a normal form, however normal forms tend to produce very large game trees. The sequence form is an efficient method of representing extensive form games(Koller et al. 1996). This representation is

described as a linear-sized strategic description of the game. It decomposes players strategies into sequences of actions and probabilities of realizing those sequences.

In sequence form games, for every player, $i \in N$, each of their information states $u^i \in U^i$ uniquely define a **sequence** σ_{u^i} of actions that a player must take in order to reach that information state. These sequences are then mapped to realization probabilities through what is called a realization plan, denoted by x. When two or more strategies have the same realization plan these strategies are considered to be realization equivalent(Von Stengel 1996). This can apply across different types of strategies, for example, extensive-form, behavioral strategies and normal-form, mixed strategies(Kuhn 1953).

2.2.2 Nash Equilibria

A Nash's equilibrium is a state in which each player in a game has chosen a strategy and none of the players can benefit from changing their strategies, if the other player's strategies remain unchanged. As such, if a strategy is developed that induces a Nash's equilibrium our strategy can no longer be exploited.

In the context of extensive form games the concept of best responses is related to that of Nash's equilibria. If the opponent's strategies are denoted by π^{-i} then the set of best responses are denoted by $BR^i(\pi^{-i})$. Note that for a strategy profile π such that $\pi^i \in BR^i(\pi^{-i})$ for every $i \in N$ (i.e for every player) then that game constitutes a Nash equilibrium(Heinrich 2017). Heinrich also discusses the concept of ε-best responses. This is the set of strategies that are within a certain tolerance ε of the best responses. As such an ε-Nash equilibrium can be defined as a strategy profile π such that $\pi^i \in BR^i_\varepsilon(\pi^{-i})$ for all $i \in N$.

2.2.3 Fictitious Play

Fictitious play(FP) is a game-theoretic model of learning through self-play. At each iteration, players choose the best responses to their opponents average strategies(Heinrich 2017). These strategies converge to Nash equilibria in certain classes of games, including two-player, zero-sum games.

Generalised weakened fictitious self-play (GWFSP) is a method that is built on FP but allows for approximations in players strategies(Leslie & Collins 2006). Thus it is more suitable for machine learning. GWFSP allows for a certain error at each iteration of the algorithm and relies on the fact that this error rate will tend towards zero as time progresses. In the research done by Leslie and Collins normal form games were studied. There has also been research done into the applicability of FP to extensive form games, however, before(Heinrich & Silver 2016) there was no method shown to converge for imperfect information games such as poker. In later sections, we will discuss how Heinrich utilised GWFSP as a basis for neural fictitious self-ply, a method that has shown success in games of imperfect information.

2.3 Texas Hold'em

Although it was decided that a full-scale implementation for Texas hold'em would be beyond the scope of this project, this was the target game at the inception of the project. As such the mechanics of this game will now be discussed. The reader may choose to skip to section 2.3.4 in order to only learn about the game of Leduc Hold'em.

Texas hold'em is one variant in a family of games called poker. Poker is a group of card games that combine gambling, strategy, and skill. All poker variants have three core similarities. There is betting involved, there is imperfect information (ie cards remain hidden until the end of a hand) and the winner is determined by combinations of cards.

2.3.1 Game Structure

Texas hold'em consists of four betting rounds. Initially, each player is dealt two private cards. These remain face down and only the person who received these cards may view them. In the next three rounds, five public cards are dealt face up on the table. The second round of dealing is called the flop, where three public cards are dealt. The third round is called the turn where one additional public card is dealt. Finally, in the fourth round, another public card is dealt which is called the river.

At each round, after the cards are dealt, the players are given the opportunity to take a number of betting related actions. The next section outlines the permitted actions.

In order for players to be incentivized to continue playing in a wider array of situations, blinds are required. Blinds are a mandatory bet that must be posted by two of the players present at the game. These two bets are called the big blind and the small blind, the big blind being twice that of the small blind. As hands are played the big and small blinds are posted by different players in order to distribute the cost fairly.

The big and small blinds are the first two bets that contribute to what's called the pot. The pot is the collection of all of the current chips bet by the players. When a player wins a hand then what they receive in return is the pot.

The final structural component of the game is player stacks. Each player will start the game with a certain amount of chips. If a player wins a pot then all of the chips in the pot are transferred to the winner's stack.

2.3.2 Actions

As mentioned in the previous section, after cards are dealt players are permitted to take a number of actions. If a player is the first to act the they may either check or bet a chosen amount. If a player is not first to act and the previous player has made a bet then they may choose to either fold and

forfeit the pot, call the bet by adding the same amount to the pot, or raise by adding the amount previously bet plus some additional chips. Players can go back and forth with bets until they run out of chips in which case they are considered to be "all in".

2.3.3 Hand Values

In poker, the best 5 cards available to the player can be played. This means any combination of his own private cards and the public cards can be used. There are 10 major poker hands. These are listed below in ascending order of value:

1. **High Card:** None of the higher hand values achieved, highest card plays.
2. **Pair:** Any two cards of the same rank.
3. **Two Pair:** Two different pairs.
4. **Three of a kind:** Three cards of the same rank.
5. **Straight:** Five cards in a sequence.
6. **Flush:** Five cards of the same suit.
7. **Full House:** Three of a kind with a pair.
8. **Straight Flush:** Five cards in sequence, all of the same suit.
9. **Royal Flush:** A, K, Q, J, 10 - all of the same suit.

2.3.4 Leduc Hold'em

Leduc Hold'em is a simplified version of Hold'em that was first introduced in(Southey et al. 2012). In Leduc Hold'em the deck is reduced to six cards with two suits and three ranks in each suit. Rather than four rounds there are only two. In the first round, a single private card is dealt to each player.

In the second round, a single board card is revealed. In the first round, both players have a mandatory bet of one and a raise of two is allowed. In the second round, players can raise by four. Each round allows for at most two bets.

2.4 Variations of MCTS Applied to Poker

As mentioned in section 2.1.7, MCTS has been shown to be a very powerful method for solving large perfect-information games such as Go. However if this algorithm is to be applied to a imperfect information game like poker then a number of modifications must be applied. Such an approach was outlined in(Heinrich 2017) in which a variation of MCTS called smooth UCT was implemented to tackle both leduc hold'em and limit hold'em.

2.4.1 Extensive Form MCTS

One of the subtleties of poker is that player information is asymmetric. In other words, each player has access to their own private card information but not to their opponent's private cards. This means that the search tree cannot be a single, collective entity(Heinrich 2017), rather two search trees must be available to accommodate self-play. The method that can be used to accomplish this goal is the creation of an information function $I^i(s)$, a concept from game theory(see section 2.2.1). This function will return the information state u^i of the current player (i) given the current state s from the full game tree. Note that the full game tree will have separate nodes for any variation in either player's cards. However, player 1's game tree will not have separate nodes where the only differentiating factor is player 2's private cards(Johanson et al. 2011) due to the fact that this information is not available. For example, let's say that state s is in the full game tree and contains the information that player 1 has an ace, player 2 has a king and there has been a single bet so far in the game. If this state is passed to player 1's information function the returned information state,u^i, will only contain

the information that player 1 has an ace and there has been a single bet thus

far. Algorithm 1 shows Heinrich's extensive form MCTS .

Algorithm 1: Extensive Form Monte Carlo Tree Search

```
1  Function SEARCH():
2      while Within computational budget do
3          SIMULATE(s_0)
4      end
5      return π_tree
6  Function ROLLOUT(s):
7      a ~ π_rollout(s)
8      s' ~ G(s, a)
9      return SIMULATE(s')
10 Function SIMULATE(s):
11     if ISTERMINAL(s) then
12         return r ~ R(s)
13     end
14     i = PLAYER(s)
15     if OUT-OF-TREE(i) then
16         return ROLLOUT(s)
17     end
18     u^i = I^i(s)
19     if u^i ∉ T^i then
20         EXPANDTREE(T^i,u^i)
21         a ~ π_rollout(s)
22         OUT-OF-TREE(i) ← true
23     else
24         a = SELECT(u^i)
25     end
26     s' ~ G(s, a)
27     r ← SIMULATE(s')
28     UPDATE(u^i,a,r^i)
29     return r
```

Aside from the differences mentioned, extensive form MCTS remains essentially the same as generic MCTS . On lines 6 to 9, the rollout function is listed. Here an action is sampled using the rollout policy which is then used to generate a subsequent state through invocation of the state transition function G.

The SIMULATE function is the core recursive driver behind the MCTS algorithm. Initially, we check if we have reached a terminal state if so the reward function, R is utilised to calculate this reward which is then returned. Next, on line 14, the PLAYER function is used to determine which player is next to move. Then we check a data structure called OUT-OF-TREE that keeps track of which player has left the scope of their search tree in the current episode(Heinrich 2017). If we are outside the scope of the tree then we utilise the ROLLOUT function. Lines 19 to 22 handle the case in which the current information state, u^i is not in player i's tree, T^i. In this case, this node is added to the tree using the EXPANDTREE function, and we begin to use the rollout policy to generate behavior. On line 24 we have the contrary case in which we continue to use the tree policy by invoking the SELECT function. On lines 26 to 29, the next state is generated after which the SIMULATE function is recursively called. When this recursion ends, the UPDATE function is called in order to update the values and visitation counts of u^i and it's child node.

2.4.2 UCT

In order to understand smooth UCT we must first explain the meaning of UCT . The abbreviation UCT refers to upper confidence bound(UCB) applied to trees. This means that in the action selection portion of the algorithm a UCB approach is taken, where less explored nodes in the tree are given a positive bias in value. This means that unexplored nodes will be explored which ensures that the best actions at each position in the tree are discovered. The value of an action is denoted as follows:

$$Q(u^i, a) + c\sqrt{\log N(u^i)/N(u^i, a)} \tag{2.10}$$

In this expression the Q function denotes the current value estimates for each action a in information state u^i. $N(u^i)$ and $N(u^i, a)$ denote the visitation count of the information state u^i and the subsequent information state after action a has been taken. The visitation count of a state is simply the number of times the algorithm has passed through that state. Additionally, c is the exploration constant. This constant can be used to control the level of exploration during the execution of the algorithm, with a value of 0 resulting in a purely greedy approach. This extension of the extensive form MCTS algorithm is shown in Algorithm 2.

Algorithm 2: UCT

SEARCH, SIMULATE, and ROLLOUT as in Algorithm 1

Function `SElECT()`**:**

  return $argmax_a Q(u^i, a) + c\sqrt{\dfrac{\log N(u^i)}{N(u^i, a)}}$

Function `UPDATE(`u^i`,`a`,`r^i`)`**:**

  $N(u^i) \leftarrow N(u^i) + 1$

  $N(u^i, a) \leftarrow N(u^i, a) + 1$

  $Q(u^i, a) \leftarrow Q(u^i, a) + \dfrac{r^i - Q(u^i, a)}{N(u^i, a)}$

2.4.3 Smooth UCT

Smooth UCT is a modification of UCT that is inspired by fictitious play(Heinrich 2017). As described in the last section the action selection in UCT is purely deterministic. Smooth UCT changes this by utilizing the average strategy a certain proportion of the time. In order to calculate the average strategy for a particular state, we create a probability distribution based on the visitation counts of the actions available at that state. For example, we could have an information state u^i that has been visited 100 times and has three available

actions, a_1, a_2, a_3. Then it would be possible that a_1 has been visited 50 times, a_2 35 times and a_3 15 times. According to the average strategy we would then select a_1 50% of the time, a_2 35% of the time and a_3 15% of the time. This process can be seen in line 9 and 10 of Algorithm 3. Here, for each available action a at information state u^i we set the probability of selecting that action to be the visitation count of the child node ($N(u^i, a)$) divided by the visitation count of the parent ($N(u^i)$). In order to determine when the average strategy should be used compared to the UCB approach Heinrich describes a sequence η_k. This sequence decays to 0 as $\lim_{k\to\infty}$ and is expressed as follows:

$$\eta_k = max(\gamma, (1 + d * \sqrt{N_k})^{-1}) \tag{2.11}$$

Here N_k is the total number of plays, γ is a lower limit on η_k and d is a constant that parametrises the rate of decay. This is calculated on line 4 of Algorithm 3.

Algorithm 3: Smooth UCT

1 SEARCH, SIMULATE and ROLLOUT as in Algorithm 1
2 UPDATE as in Algorithm 2
3 **Function** `SELECT()`**:**
4 $\quad \eta \leftarrow max\left(\gamma, \left(1 + d * \sqrt{N_k}\right)^{-1}\right)$
5 $\quad z \sim U[0, 1]$
6 $\quad$ **if** $z < \eta$ **then**
7 $\quad\quad$ **return** $argmax_a Q(u^i, a) + c\sqrt{\dfrac{\log N(u^i)}{N(u^i, a)}}$
8 $\quad$ **else**
9 $\quad\quad \forall_a \in A(u^i) : p(a) \leftarrow \dfrac{N(u^i, a)}{N(u^i)}$
10 $\quad\quad$ **return** $a \sim p$
11 $\quad$ **end**
12 $\quad$ **return** π_{tree}

In chapter 3 we will take this abstract algorithm and document how it was translated to python code.

2.5 Other Approaches

Although it has already been stated that there will be a focus on MCTS, it's worth discussing some other notable methods for creating poker agents before proceeding. Currently, the premier method for tackling full-scale texas hold'em is counterfactual regret minimization. This is the method that has dominated the Annual Computer Poker Competition(ACPC) for the last number of years, however recently some new methods have been outlined which look promising and thus will be discussed as well.

2.5.1 Counterfactual Regret Minimization

Counterfactual regret minimization(CFR) is a method for finding approximate Nash equilibria in imperfect information games and was first outlined in(Zinkevich et al. 2008). Regret is a measure of the difference in utility between following some strategy σ compared to another strategy. Zinkevich introduced counterfactual regret which is regret applied to a single information set ie a single extensive form game state. It was found that by calculating and minimizing regret on an individual state basis that there were performance benefits as well as an improvement in accuracy of the calculated approximate Nash equilibria.

One implementation of CFR is outlined in(Jackson 2013). This paper outlines the implementation used for Slumbot, the 2018 ACPC champion in no-limit hold'em. Jackson describes his use of CFR in order to generate a static strategy that approximates a Nash equilibrium. At each iteration a strategy would be computed and when the process had ended an average of these strategies would be used. Jackson also mentions using an abstraction of the game in order to reduce no-limit's massive state space (roughly 10^{164} with stack sizes of 100 big blinds)(Johanson 2013). This is done through

techniques like grouping similar hand values into buckets and treating them as strategically equal or splitting the game up into its individual rounds and solving the rounds separately. These techniques allow for a dramatic reduction in state space and if handled correctly, should allow for a strategy that can transfer to the full version of the game successfully.

2.5.2 Neural Fictitious Self-Play

As mentioned earlier Fictitious Play(FP) is a game-theoretic model that allows learning through self-play. This model was then extended a number of times until generalised weakened fictitious play(GWFP) was developed as a method of self-play that was applicable to machine learning techniques. Heinrich then utilised this method to develop Fictitious Self-Play(FSP)Heinrich et al. (2015). This is a machine learning framework that implements fictitious play. This method was shown to successfully generate approximate Nash's equilibria in imperfect information games such as Leduc Hold'em.

Neural Fictitious Self-Play(NFSP) develops upon these methods through the use of neural networks in order to facilitate the larger state spaces of games such as limit texas hold'em. This method was outlined inHeinrich & Silver (2016).

Algorithm

NSFP agents learn through self-play ie interaction with other instances of the agent. As this process is happening the agents store experience of their play in two memories named M_{RL} and M_{SL}. These memories are subsequently used as data sources for the reinforcement learning and supervised classification portions of the algorithm respectively.

This algorithm trains two neural networks which are in-turn used to generate two strategies. The first neural network $Q(s, a|\theta^Q)$ is trained to predict action values from data in M_{RL} using off-policy reinforcement learning. The second neural network $\Pi(s, a|\theta^\Pi)$ is used to imitate it's own past best-response behavior using supervised classification on the data gathered

in M_{SL}. The resultant strategies are the best-response strategy $\beta = \epsilon - greedy(Q)$, which uses the previously mentioned ϵ-greedy method for balancing exploration and exploitation, and the average strategy $\pi = \Pi$.

Neural Fictitious Self-Play (NFSP) with fitted Q-learning

Initialize game Γ and execute an agent via RUNAGENT for each player in the game
function RUNAGENT(Γ)
  Initialize replay memories $\mathcal{M}_{RL}$ (circular buffer) and $\mathcal{M}_{SL}$ (reservoir)
  Initialize average-policy network $\Pi(s, a \,|\, \theta^{\Pi})$ with random parameters θ^{Π}
  Initialize action-value network $Q(s, a \,|\, \theta^{Q})$ with random parameters θ^{Q}
  Initialize target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
  Initialize anticipatory parameter η
  for each episode **do**
    Set policy $\sigma \leftarrow \begin{cases} \epsilon\text{-greedy}\,(Q)\,, & \text{with probability } \eta \\ \Pi, & \text{with probability } 1-\eta \end{cases}$
    Observe initial information state s_1 and reward r_1
    for $t = 1, T$ **do**
      Sample action a_t from policy σ
      Execute action a_t in game and observe reward r_{t+1} and next information state s_{t+1}
      Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in reinforcement learning memory $\mathcal{M}_{RL}$
      if agent follows best response policy $\sigma = \epsilon$-greedy (Q) **then**
        Store behaviour tuple (s_t, a_t) in supervised learning memory $\mathcal{M}_{SL}$
      end if
      Update θ^{Π} with stochastic gradient descent on loss
        $\mathcal{L}(\theta^{\Pi}) = \mathbb{E}_{(s,a)\sim\mathcal{M}_{SL}}\left[-\log \Pi(s, a \,|\, \theta^{\Pi})\right]$
      Update θ^{Q} with stochastic gradient descent on loss
        $\mathcal{L}\left(\theta^{Q}\right) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{M}_{RL}}\left[\left(r + \max_{a'} Q(s', a' \,|\, \theta^{Q'}) - Q(s, a \,|\, \theta^{Q})\right)^2\right]$
      Periodically update target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
    end for
  end for
end function

Figure 2.5: Neural Fictitious Self-Play(Heinrich & Silver 2016)

As shown in figure 2.5 at each iteration the policy is selected first. Next an action a_t is sampled from that policy and by taking that action we generate a reward and subsequent state. This information is stored in M_{RL} and conditionally in M_{SL} based on the policy selected. After this stochastic gra-

dient descent is used to update θ^{Π} and θ^{Q} who's new values are periodically applied to our neural networks. Over time the action value and average-policy networks will improve their accuracy as each player gains experience and learns. This culminates in an approximate Nash equilibrium as players develop better and better strategies.

Convergence and Empirical Results

This algorithm was applied to both Leduc Hold'em and Limit Texas Hold'em with variation in a number of parameters such as the number of hidden layers and neurons in the neural networks.

The best results achieved for Leduc Hold'em used a single hidden layer neural network with 128 hidden neurons. This instantiation of the algorithm developed a strategy reached exploitability levels of less than .1, with a starting value of 1.5.

The limit Texas hold'em instantiation of the algorithm was tested against the top 3 competitors of the 2014 ACPC using win rate as the evaluation metric in milli big-blinds per hand (mbb/h). This saw a trained win rate of between -13 and -52mbb/h against these agents with the initial win rate being roughly -700mbb/h.

Chapter 3

Implementation

In this chapter the details of the product developed will be discussed. This includes the implementation of the MCTS algorithm, our metric computation and the development of our prototype game for demonstration purposes.

3.1 Smooth UCT Algorithm

The core component of our implementation is the smooth UCT algorithm, as outlined in section 2.4. As such, in this section, the code corresponding to this algorithm will be outlined.

3.1.1 Tree Representation and Utility Functions

This subsection seeks to provide background on the data structures implemented and some of the functions heavily utilised in the following chapter. These are not core elements of the algorithm and thus **the reader may choose to skip to section 3.1.2**.

Histories

As mentioned in section 2.1.2, when dealing with a POMDP, we must use histories to represent states. These histories consist of a sequence of actions

and observations(Silver & Veness 2010). When practically applied to the game of Leduc Hold'em this means that we must store all of the cards dealt and the actions taken as a part of each history. In order to store history information in a simple, parsable manner it was decided that strings would be used. Their primary components would be cards and actions. In order to simplify the data involved abbreviations were used for both cards and actions. Below is the list of abbreviations for cards:

- **Ah** - Ace of Hearts
- **As** - Ace of Spades
- **Kh** - King of Hearts
- **Ks** - King of Spades
- **Qh** - Queen of Hearts
- **Qs** - Ace of Spades

Below is the list of abbreviations for actions:

- **f** - Fold
- **c** - Call
- **b** - Bet
- **r** - Raise

The final component required to represent a history was a prefix. The prefix corresponds to the player that is first to play in the game and facilitates the computation of the next player to take an action given any history. In our case, it could be either 1 or -1. The table below gives a full breakdown of an example history.

Prefix	**Private Cards**	**Round 1 Actions**	**Public Cards**	**Round 2 Actions**
1	AhQh	bc	Ks	brc

Game Tree

The tree data structure consists of two primary components: histories and nodes. A python dictionary was used with the keys being histories and the values being nodes. Nodes were defined as simple data structures that contained a visitation count, a value, a parent and a set of children. Both the parent and the children were histories that also existed as keys in the tree dictionary thus facilitating $\mathcal{O}(1)$ lookup time.

```
class PoNode:

    def __init__(self):
        self.visitation_count = 1
        self.value = 0
        self.parent = ""
        self.children = set()

# Example instantiation.
# In our implementation the root node was was an empty string.
tree = {"": PoNode()}
```

Strategies

In the rest of the chapter different types of strategies are discussed. Specifically, we refer to deterministic and stochastic strategies.

When referring to deterministic strategies we mean strategies that have a single action associated with each state or history. This means that whenever that history is reached the same action will be selected each time. In the implementation, these strategies are derived by analysing the values of nodes in the game tree. The action that is taken at any given history is the action that will lead us to the child whose value is the highest.

Stochastic strategies are strategies that provide a probability distribution across the available actions at a given state or history. This means that each

action may be taken at a particular history but, there is a different probability of taking each action. These strategies are derived as a direct result of the average strategy used by the agent. In other words, the visitation counts of nodes during the MCTS training. Due to the fact that MCTS samples states at a biased rate based on the expected value of the state we can use this information to induce stochastic strategies. This process is outlined in more detail in section 2.4.3.

Utility Functions

The first utility function is the util.terminal function. This function corresponds to the TERMINAL function listed in Algorithm 1 and takes a history as an argument returning a boolean value. For example, if a player has folded then the return will be true.

The second utility function is the util.player function. This corresponds to the PLAYER function listed in Algorithm 1, again taking a history as it's sole argument. Depending on who is next to act it will return 1(player 1), 0(dealer) or -1(player 2).

The final utility function that will be discussed is the util.information_function. This function corresponds to the information function, I, in Algorithm 1. It is parameterised by a player and a full game history. The return value is that same history without the opponent's private card embedded in the history. For example passing parameters ("1AhQhbc",1) would return "1Ahbc". Note that the second card "Qh" has been stripped from the returned history.

3.1.2 Selection

The python implementation for Smooth UCT action selection was a relatively close approximation of the algorithmic SELECT function as discussed in section 2.4. The first distinction that should be made is that in section 2.4 states and information states were mentioned when discussing the Smooth UCT algorithm. In this chapter histories and player histories will be discussed as parallel concepts in the implementation. Another notable

difference is that instead of storing the history values and visitation counts in separate Q and N functions these functions are embedded within the tree data structure. This means that when the functions at line 11 and 13 are called these functions directly correspond to lines 7 and 9 in Algorithm 3. Due to the fact that these functions are relatively large despite serving a trivial purpose they have been omitted for brevity.

```
1  def select(history):
2      player = util.player(history)
3      player_history = util.information_function(history, player)
4      tree = get_tree(player)
5
6      eta_sub_expression = math.pow(1 + d *
           math.sqrt(tree[player_history].visitation_count), -1)
7      eta = max((GAMMA, ETA_ZERO * eta_sub_expression))
8
9      z = random.uniform(0, 1)
10     if z < eta:
11         return get_best_action_ucb(player_history, tree)
12     else:
13         return get_action_avg_strategy(player_history, tree)
```

3.1.3 Expansion

Due to the fact that(Heinrich 2017) did not discuss the details of his EXPANDTREE function (see Algorithm 1), it was decided that the expansion method detailed in(Silver & Veness 2010) would be used. This was the original paper that outlined the method of applying MCTS to POMDPs so it seemed an adequate alternative. Silver's method stated that all available child nodes would be appended as children to the node to be expanded. The resulting code is listed below.

```
1  def expand(tree, history):
2      player = util.player(history))
3      player_history = util.information_function(history, player)
4
5      if player_history not in tree:
6          tree[player_history] = potree.PoNode()
7
8      for action in util.get_available_actions(player_history,
           player=player):
9          new_history = player_history + action
10         if new_history not in tree:
11             tree[new_history] = potree.PoNode()
12         tree[player_history].children.add(new_history)
```

3.1.4 Simulation

In this section, we will discuss the python equivalents to the ROLLOUT and SIMULATE functions from Algorithm 1.

In the python implementation, the rollout policy, $\pi_{rollout}$, is simply a random policy(line 2). This means that each action that is available in that state/history has the same probability of being selected. Furthermore, the state transition function, G, is replaced by line 3 below. This is because the child of a history will always be an action appended to that history.

```
1  def rollout(history):
2      action = random.choice(util.get_available_actions(history))
3      new_history = history + action
4      return simulate(new_history)
```

The SIMULATE function, when ported to python code had a number of modifications. First, a tree function is introduced. This function has the purpose of returning the tree relevant to the current player.

It was also found that an additional condition had to be added for the if statement at line 11. This condition ensured that the player history had children. If it did not then the else block would be executed and those children would be added. During the initial implementation of the algorithm, it was found that the tree would become disconnected if this condition was not present, with every second layer of the tree having no children.

The final divergence from Algorithm 1's SIMULATE function is the conditional on line 14. In the case of our implementation, the player can be either of the active players or the dealer. If the current player is the dealer then we do not want to call the expand function, nor set the OUT-OF-TREE property to true.

```
1  def simulate(history):
2      if util.is_terminal(history):
3          return handle_terminal_state(history)
4
5      player = util.player(history)
6      if out_of_tree[player]:
7          return rollout(history)
8
9      player_history = util.information_function(history, player)
10     player_tree = tree(player)
11     if player_history in player_tree and
           player_tree[player_history].children:
12         action = select(history)
13     else:
14         if player != DEALER:
15             expand(player_tree, history, player)
16             out_of_tree[player] = True
17         action = random.choice(util.get_available_actions(history))
18
19     new_history = history + action
```

```
    running_reward = evaluator.calculate_reward_full_info(history) + \
                         discount_factor * simulate(new_history)
    update_player_tree(history, action, player, running_reward)

    return running_reward
```

3.1.5 Tree Update

The tree update function is essentially identical to the function listed in Algorithm 2 with the only difference being that the Q and N are embedded in the game tree. The code is listed below.

```
def update(tree, history, new_history, running_reward):
    tree[history].visitation_count += 1
    tree[new_history].visitation_count += 1
    tree[new_history].value += (running_reward -
        tree[new_history].value) / tree[new_history].visitation_count
```

3.2 Exploitability Computation

In order to evaluate the performance of our agent exploitability was utilised as the primary metric. Exploitability is a measure of how well our agent would fare against an opponent responding optimally to our strategy, ie an opponent using a best response strategy. Exploitability is related to the concept of Nash equilibria in that a Nash equilibrium is induced by a strategy that cannot be exploited. In this section we will discuss the process of calculating exploitability step by step and giving coding examples. In order to do so we will refer to three player's that are related to the process. These are the best response(BR) player, the MCTS player and the chance player. The BR player is the player who's strategy we are attempting to establish through computation of the best response strategy. The MCTS player is the player

whose strategy has already been established through execution of the MCTS algorithm. The chance player is the player who is responsible for the chance actions of the game ie dealing cards.

3.2.1 Generating Best Response Tree

To calculate the exploitability of a strategy the best responses to that strategy must be determined. In other words, the best response strategy must be established.

There are two steps involved in this process. The first step involves generating a full game tree. This can be implemented as a relatively trivial recursive function in which we begin at the root of the tree and at each level add every possible child node until a terminal node is reached. It is worth noting that the tree generated is a full game tree. This means that all of the game information is represented in the tree. As a result, if we wish to view the nodes or histories in the tree from either player's perspective we must apply the information function in order to remove the opponent's private information from view. The distinction between the full game tree and the player's information set is illustrated in figure 3.1.

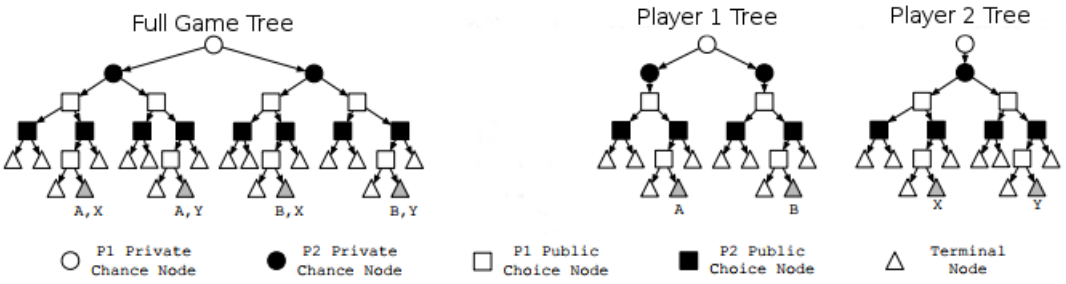

Figure 3.1: Full Game Tree vs Player Information Set - adapted from(Johanson et al. 2011)

The code is listed below.

```
1 def build_tree(history, tree):
```

```
    tree[history] = potree.PoNode()

    if not util.terminal(history):
        actions = util.get_available_actions(history)
        for action in actions:
            child = history + action
            tree[child] = potree.PoNode()
            tree[child].parent = history
            tree[history].children.add(child)
            build_tree(child, tree)

    return tree
```

The second step in generating the best response tree involves taking the MCTS player's action selections and inserting them into the full game tree(Heinrich 2017). In other words, wherever the MCTS player must take an action in the game tree, the best action is chosen based on their MCTS estimations. As such in the resultant tree, the MCTS player's decision nodes will have a single child node. This is essentially a pruning process in which the child nodes that the MCTS player has not selected in their strategy are removed. The code is listed below.

```
def apply_mcts_strategy(mcts_tree, best_response_tree, current_history):
    if current_history not in best_response_tree:
        return best_response_tree

    # Have we reached a decision node for the MCTS player?
    if util.player(current_history) == 1:
        player_history = util.information_function(current_history, 1)
        action = util.get_best_action(mcts_tree, player_history)
        best_response_tree[current_history].children =
            set(current_history + action)
```

```
10
11      for history in best_response_tree[current_history].children:
12          apply_mcts_strategy(mcts_tree, best_response_tree, history)
13
14      return best_response_tree
```

As shown in the algorithm the function takes the tree that was evaluated through execution of the MCTS algorithm, the best response tree, and the current history being processed. In a recursive manner, we traverse down through the tree applying the MCTS player's strategy to each child node until we leave the tree. The player function is used to check if a decision node for the MCTS player has been reached. If so we first apply the information function to remove the opponent's private information from current history. Next, the best action according to the MCTS player is obtained. This action is then concatenated with the original history producing our desired child node. This child node is then assigned as the single child of the current history being processed. Figure 3.2 gives a visual representation of this pruning process. In this diagram, we assume that in the MCTS tree, betting has a higher associated value than folding.

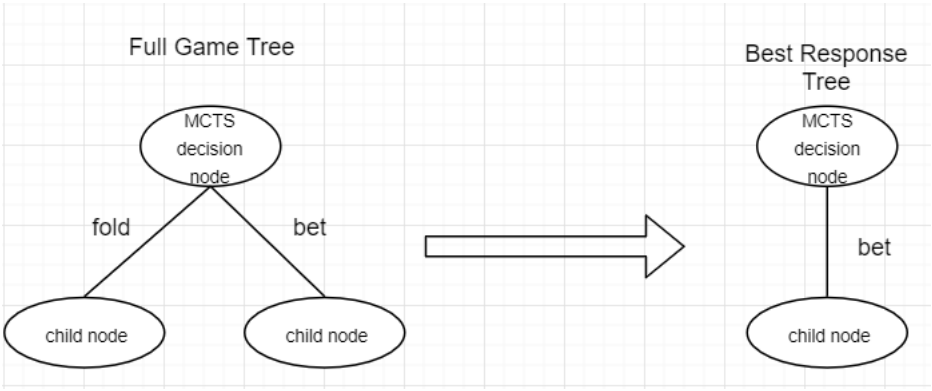

Figure 3.2: Generation of Best Response Tree

3.2.2 Terminal Node Evaluation

The next step in this process is to evaluate the terminal nodes in the best response tree. This process seems trivial on immediate inspection. However, there is one important factor that must be considered. That is that the BR player should not be allowed to choose different actions based on the private information of the opponent. This problem is avoided by calculating the average value of all histories that are equivalent in the eyes of the BR player and applying this value to the terminal node that we are evaluating. As an example, if the BR player has a king then the MCTS player may have a queen, a king, or an ace. Each of these possibilities would result in a different value for the terminal. Due to the fact that the BR player cannot see this information, they should not be able to vary their behavior based on it. As such we simply take all of these possibilities into account and assign the average value for the terminal. The code for this step is listed below.

```
1 def evaluate_terminals(best_response_tree, terminals):
2     for history in terminals:
3         if history.endswith("f"):
4             best_response_tree[history].value =
                  gevaluator.calculate_reward(history)
5         else:
6             eq_nodes =
                  util.get_information_equivalent_nodes(best_response_tree,
                  history, -1)
7             avg = evaluator.average_reward(eq_nodes)
8             best_response_tree[history].value = avg
```

It is worth noting at this stage that although the self-play smooth UCT algorithm is utilising stochastic strategies in training, the strategy evaluation step is evaluating a deterministic strategy. This can be seen in the "apply_mcts_strategy" function listed above, where every child aside from the highest valued child is being removed from the tree. If the MCTS player's

strategy was to be treated as a stochastic strategy then all child nodes would be retained at this step. Probabilities would be associated with each child depending on how often they had been visited during training.

When evaluating the exploitability of stochastic strategies one modification to this step of the algorithm would be made. A vector of reach probabilities(Johanson et al. 2011) would be calculated and these probabilities would be applied across "information equivalent" terminal nodes for the BR player. In other words, instead of simply averaging the return of these terminal nodes, they would be weighted. For example, if R is the reward function p is a vector of reach probabilities and h^i is a vector of information equivalent nodes then the stochastic value would be:

$$v(h) = p_0 * R(h_0) + p_1 * R(h_1) + \cdots + p_{n-1} * R(h_{n-1}) \tag{3.1}$$

In contrast the deterministic evaluation is as follows:

$$v(h) = \frac{R(h_0) + R(h_1) + \cdots + R(h_{n-1})}{n} \tag{3.2}$$

3.2.3 Propagating Node Values

The final step of this process involves the propagation of values back up the tree, with the highest child value being propagated when a decision node for the BR player is reached. The average child value is propagated for chance nodes. Since the MCTS player will only have a single child node for each history, that node's value is propagated each time. The code is listed below.

```
1 def propagate_rewards_recursive(best_response_tree, history):
2     parent = best_response_tree[history].parent
3
4     if util.player(parent) == 0:
5         value_to_propagate =
              util.get_average_child_value(best_response_tree, parent)
```

```
6       elif util.player(parent) == 1:
7           value_to_propagate =
                best_response_tree[next(iter(best_response_tree[parent].children))].value
8       else:
9           best_sibling = util.get_best_child(best_response_tree, parent,
                -1)
10          value_to_propagate = best_response_tree[best_sibling].value
11
12      best_response_tree[parent].value = value_to_propagate
13      propagate_rewards_recursive(best_response_tree, parent)
```

3.2.4 Exploitability

When the process outlined is complete the value that is propagated back to the root node is our exploitability value.

3.3 Prototype Application

In order to give visual evidence of the work done and the agent developed, a user interface(UI) was created that allowed human interaction with the trained bot. In order to facilitate the development of this interface in a short period of time, the Qt framework was used. Qt is an open source widget toolkit for creating graphical user interfaces and applications. Specifically, Qt Designer was used in order to generate a UI template. Qt Designer is a what-you-see-is-what-you-get (WYSIWYG) UI generation tool that accompanies Qt. PyQt5 was then used to generate python code from this UI template and connect the functional component of the game. In this section, the process involved in creating this prototype application will be outlined and some key code snippets will be highlighted.

In order to elicit requirements for this application, I brainstormed as well as analysing a number of online UIs that served a similar purpose to mine.

This process rendered the following functional requirements:

- Display list of available actions.
- Allow user to take action.
- Display relevant cards to player.
- Annotate the sequence of events that occur in the game.
- Display the current pot size.
- Allow player to play multiple rounds.
- Keep track of cumulative winnings across multiple rounds.

3.3.1 UI Screen

The first step towards achieving these functional requirements was to generate a UI screen. This was achieved through Qt designer using drag and drop with the end result being as shown in figure 3.3. Due to the fact that a number of images had to be displayed for the available cards in the game a Qt resource file was also created to store and locate those images.

Qt designer generates a .ui file that stores the UI template as well as a .qrc resource file. The following commands were used to convert these two files into python format in order for them to be compatible with the rest of the program:

```
$ pyuic4 -x ui.ui -o screen.py
$ pyrcc4 -o cards_resource.py cards/cards_resource.qrc
```

In order to create a visible screen the code listed below was used:

```
1 def setup_screen(self):
2     self.main_window = QMainWindow()
3     self.ui_screen = screen.Ui_MainWindow()
4     self.ui_screen.setupUi(self.main_window)
```

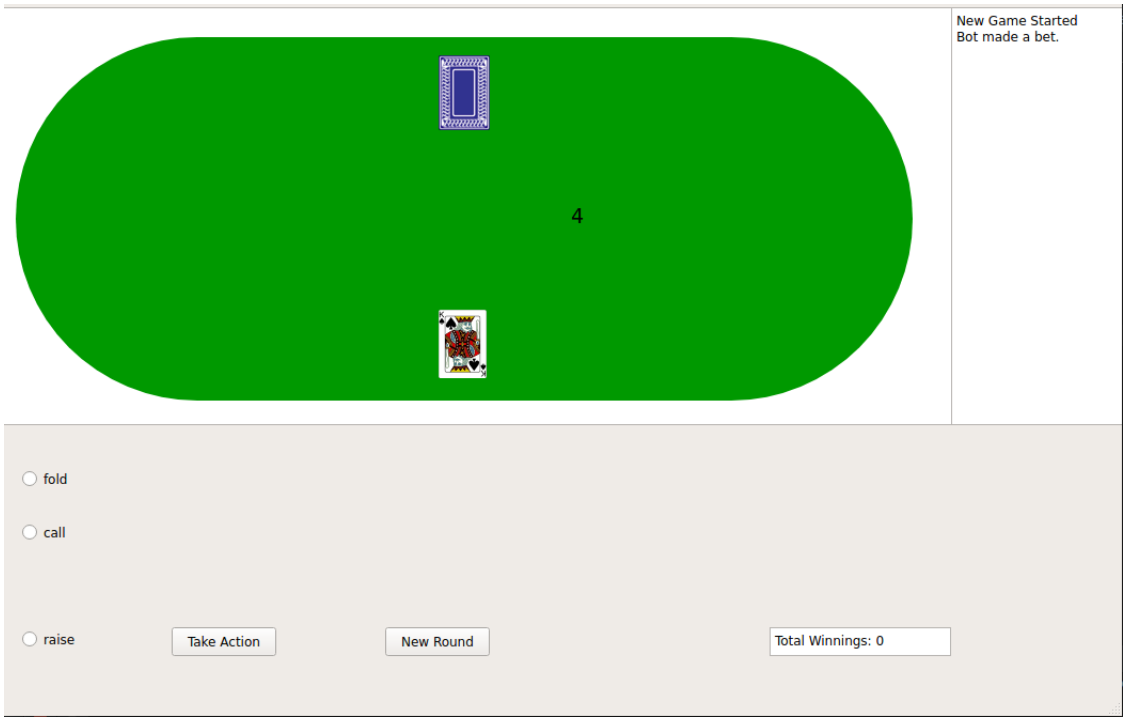

Figure 3.3: Game UI Screenshot

```
5         self.main_window.show()
6         sys.exit(self.application.exec_())
```

3.3.2 Event Handling and UI Manipulation

When the UI screen had been implemented it was then time to attach the screen to the functional aspect of the game. This was done through the creation of a python class called Controller that extends the QWidget class from the PyQt framework.

In order to handle events, callback methods were registered with the buttons that required their events to be handled. The code listed below demonstrates this process:

```
1 def setup_event_handlers(self):
2         self.ui_screen.take_action_button.clicked.connect(self.take_action)
```

```
3       self . ui_screen . new_game_button.clicked.connect(self.new_game)
```

Along with event handling the Controller class also had the responsibility of updating the UI based on the information that it gained from the game model, which will be discussed in the next section. This primarily involved calling the setText() method of a number of the screen's widgets.

3.3.3 Game Model

The third component of our game was the game model. This was instantiated through a python class named Model that had the responsibility of feeding the correct data to the Controller class at any point in the game. Due to the fact that there was already significant functionality built around the concept of histories, it was decided that the state of the game would be tracked through the use of histories. From the history, the majority of the information about the game could be deciphered. This included the available actions, the cards in play and the pot. In order to fully satisfy our functional requirements, the Model class also kept track of annotation messages that were generated based on game events, along with the cumulative winnings of the player over time. This data was returned to the Controller class in the form of a python dictionary.

The most significant method in this class was the update_game_state method. This method handled the player taking actions, the agent responding to those actions and the state being updated. The code for this method is listed below.

```
1  def update_game_state(self, action, player):
2      self.history += action
3      self.display_text += PLAYER_NAMES[player] +
           ACTION_MESSAGES[action]
4
5      # Handle the game being over
6      if util.terminal(self.history):
```

```
7            winner = evaluator.get_winner(self.history)
8            winnings = -evaluator.calculate_reward(self.history)
9            self.total_winnings += winnings
10           self.display_text += "Game over. " +
                 PLAYER_NAMES[winner] + "won: " + str(abs(winnings))
11
12       # If the next player is the bot, allow it to take its action
13       elif util.player(self.history) == 1:
14           self.update_game_state(self.agent.get_action(self.history), 1)
15
16       # If the we need cards to be dealt, deal the cards.
17       elif util.player(self.history) == 0:
18           self.pub_card =
                 random.choice(util.get_available_cards(self.history))
19           self.update_game_state(self.pub_card, 0)
```

3.3.4 Agent Representation

In the case of this prototype game, the agent is merely an instantiation of a pre-defined strategy. As such when we call agent.get_action(history) in the previous code block we are either probabilistically or deterministically selecting an action based on the aforementioned strategy. Below we have listed the code for the Agent python class. Lines 10 to 17 give the code that is executed in order to return an action if our strategy is stochastic. Line 19 gives the code to return an action if the strategy is deterministic.

```
1  class Agent:
2      def __init__(self, strategy_file):
3          self.strategy = load_strategy(strategy_file)
4
5      def get_action(self, history):
6          player_history = util.information_function(history, 1)
```

```
7
8           # Is our strategy stochastic?
9           if isinstance(self.strategy[player_history], dict):
10              candidates = []
11              probabilities = []
12
13              for key, value in self.strategy[player_history].items():
14                  candidates.append(key)
15                  probabilities.append(value)
16
17              return choice(candidates, p=probabilities)
18          else:
19              return self.strategy[player_history]
```

Chapter 4

Empirical Studies

4.1 Overview

In this chapter, we will cover a number of experiments that were conducted in order to investigate the performance of the different algorithms implemented in order to tackle Leduc Hold'em. We will begin with a simplified version of MCTS for POMDPs and will incrementally add to this method in order to see how the performance of our agent evolves. In the table below we have listed the template that we will follow when conduction these experiments.

Section	**Rationale**
Objective	This section will contain an explanation of the purpose of the experiment along with how it was carried out
Experimental Parameters	This section outlines how we parameterised the experiment.
Results	This section will detail the results acquired from the experiments conducted
Analysis	In this section we will examine our results and try to provide insight into the reasoning behind these results

4.2 Experiment 1 - UCT Versus Random Player

The first experiment conducted involved a simplified version of the algorithm outlined in(Heinrich 2017). We set an initial goal of using a random player as our benchmark opponent in order to demonstrate how this algorithm could exploit such a player's strategic inefficiencies.

4.2.1 Objective

The goal of this experiment is to implement UCT for Leduc Hold'em. The UCT agent will play against a random player and learn a strategy to exploit this player for maximal reward. Although we are interested in the results gained from playing against the random player we treat the outcome of this experiment as a baseline for our subsequent results. The reasoning for this is that the difficulty in finding a winning strategy against a random player is not high. In fact, at each point in the game that the random player can take an action it is just as likely to fold it's cards, regardless of their value, as it is to take any other action. This means that for our agent the intelligent strategy is to merely retain all but the very worst of its hands. Thus we can think of the strategy learned by this initial agent as a broad categorisation between very weak hands and all other hands. Thus we expect that the exploitability of the resultant agent will be relatively high. Our agent will not have learned all of the strategic intricacies of the game and it will not react strategically to the opponent's play. Rather, it will simply know how to beat a 'dumb' random player. This will give us a platform to build a more sophisticated agent through different mechanisms such as self-play in the subsequent experiments.

4.2.2 Experimental Parameters

For each experiment, we will list the parameters that were used to instantiate the agent. These parameters either refer to the execution time of the algorithm or are a part of the group of parameters that were outlined in

Algorithms 1, 2 and 3. In the case of each experiment, we replicate the values outlined in(Heinrich 2017). If there is a deviation from these values the reasoning behind it will be explained.

Below are listed the key parameters for the first experiment:

- **Iterations** - 500,000
- **Repetitions** - 20
- c **- exploration constant** - 18

4.2.3 Results

The first metric that was used in order to analyse the results of this experiment is cumulative reward. This is simply the sum of the output of the reward function over time. This function directly corresponds to the size of the pot won or lost in the game, thus the reward can be either positive or negative. In figure 4.1 we see the reward over time increasing. In order to obtain these results, the algorithm was run for 10,000 iterations and this process was repeated 100 times. Our cumulative rewards were then averaged at each iteration across these 100 repetitions to give the graph shown. Figure 4.2 shows the rate of increase in cumulative reward or the average reward over time. In the case of figure 4.2, we applied the UCT algorithm for 500,000 iterations and repeated this process 20 times, averaging the results.

The second metric used to produce results was exploitability. This is a measure of the reward that can be gained by playing a best response strategy against the agent. In figure 4.3 we see that the exploitability of our agent is quite high throughout, with an initial dip followed by a divergence towards 4.9.

4.2.4 Analysis

We will first analyse our results from the cumulative reward metric. As shown by figure 4.1 we see that initially there is a gradual increase in cumulative

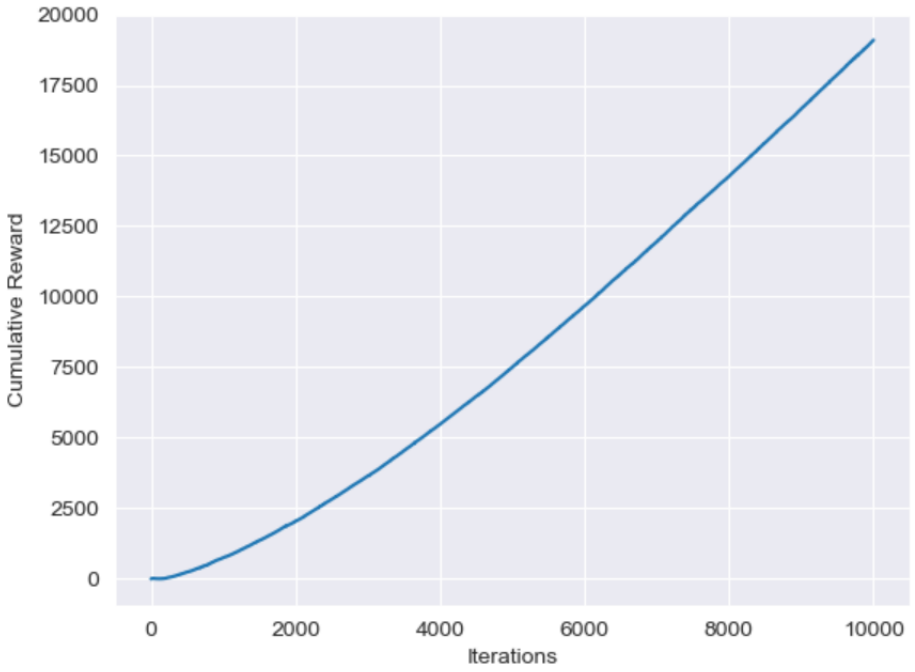

Figure 4.1: UCT cumulative reward vs random player - 10000 Iterations

reward, with the slope growing until there is a constant rate of increase. This demonstrates that during the initial phase of the algorithm we have not yet uncovered the most beneficial action selection for all states and the exploration phase is still in effect. However, by visual inspection, we can see that over time the rate of increase in cumulative reward begins to stabilise which indicates that a concrete strategy that can exploit the random play has been established. This hypothesis is further supported by figure 4.2 as we see a dramatic upswing in the rate of increase of cumulative reward followed by a leveling of the graph.

As mentioned the exploitability value throughout this figure 4.3 is relatively poor with an initial dip followed by a divergence. This is most likely explained by the fact that we are not playing against a rational player. As such our strategy is strong when it comes to maximally exploiting an irrational, random player but if we then substitute this irrational player for a rational player, the results will not be favourable.

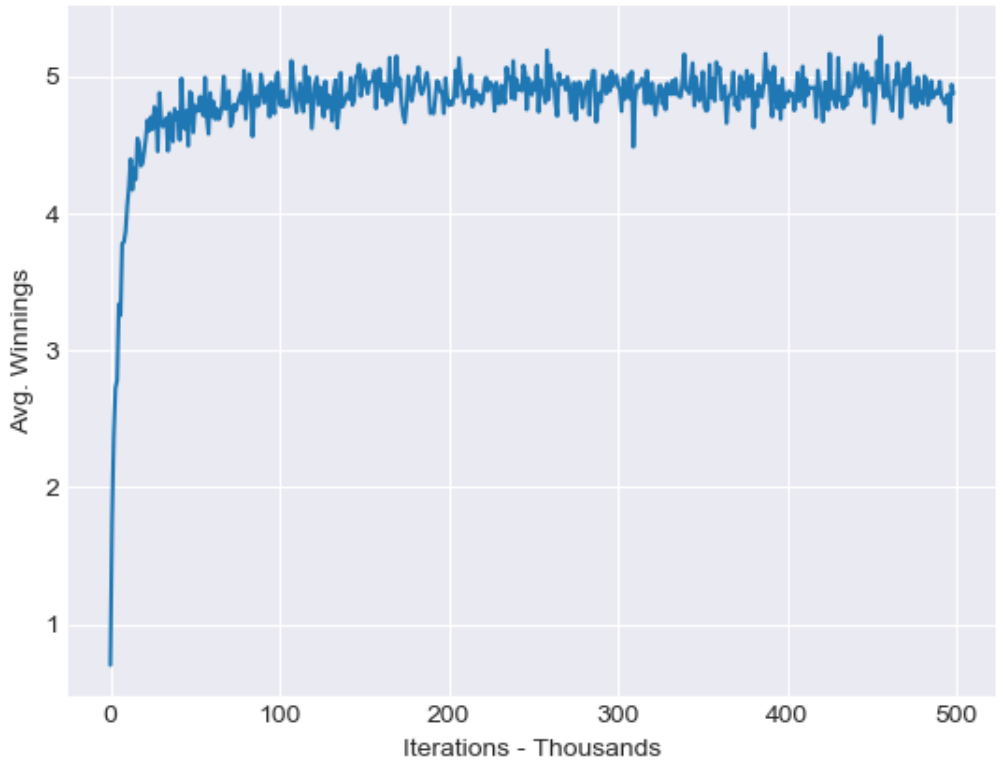

Figure 4.2: UCT average reward vs random player - 500000 Iterations

4.3 Experiment 2 - UCT Self-Play

In our second experiment, the agent was trained against itself. In other words, two instances of the agent were generated and allowed to develop strategies through self-play.

4.3.1 Objective

The objective of this experiment was to implement Heinrich's extensive form MCTS using UCB action selection (ie extensive form UCT). Here a significant improvement in the exploitability of the agent was expected. This was due to the fact that the opposing agent is rational, in that it learns through

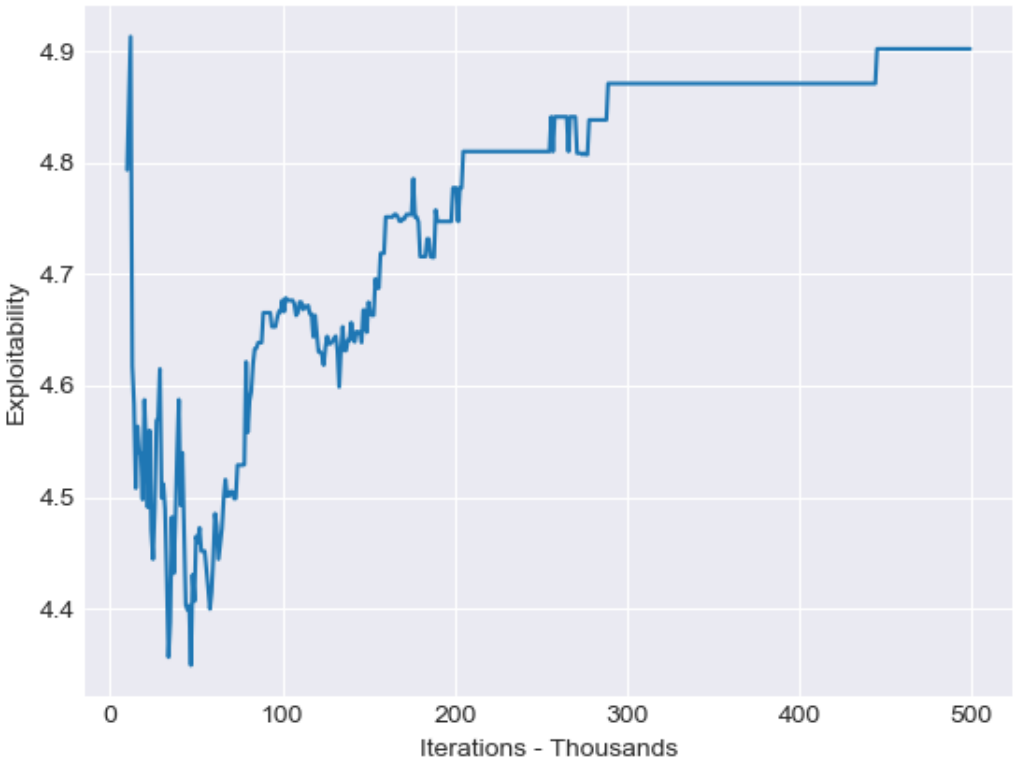

Figure 4.3: UCT exploitability vs random player - 500000 Iterations

experience.

4.3.2 Experimental Parameters

Below are listed the key parameters of the experiment

- **Iterations** - 500,000
- **Repetitions** - 10
- c **- exploration constant** - 18

4.3.3 Results

In the case of self-play both agents develop intelligent strategies. This means that we do not see any significant trends in cumulative reward or average reward over time due to the fact that neither player has an advantage. This is shown in figure 4.4. As such these metrics were disregarded and exploitability

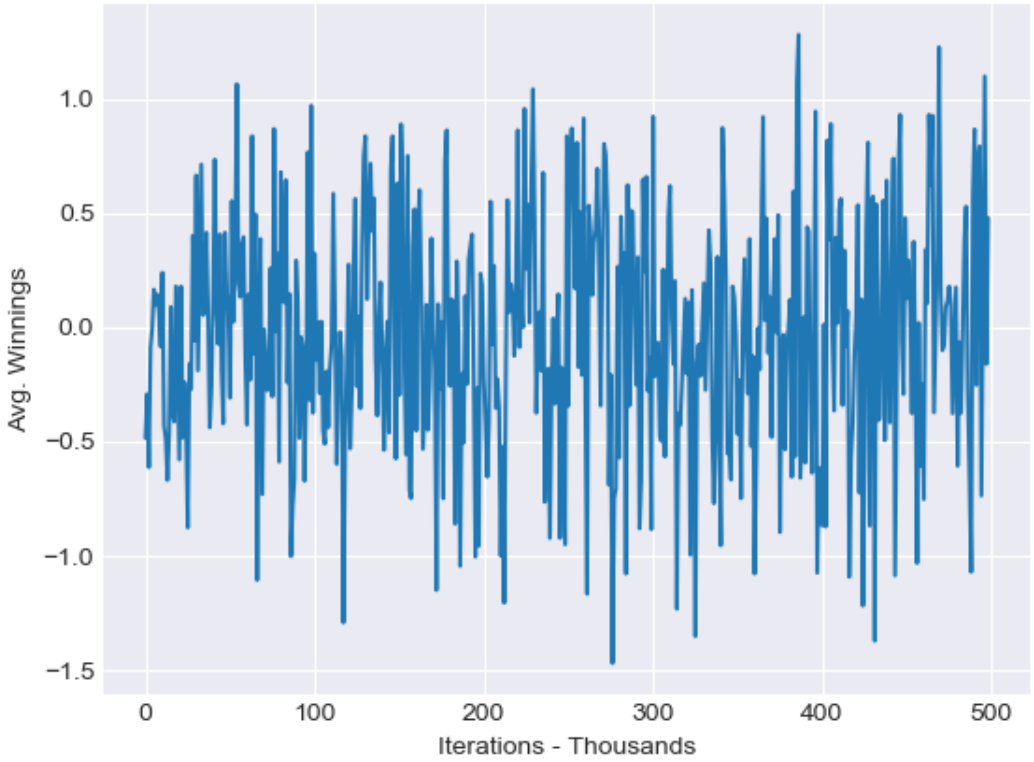

Figure 4.4: UCT average reward over time - self-play - 500000 Iterations

became the sole metric. Once again we applied the algorithm for 500,000 iterations, repeating this process 10 times and averaging the results.

In figure 4.5 it can be seen that there is a notable improvement in exploitability, with lows of 3.5 initially but over time the values begin to diverge once again back towards 4.6.

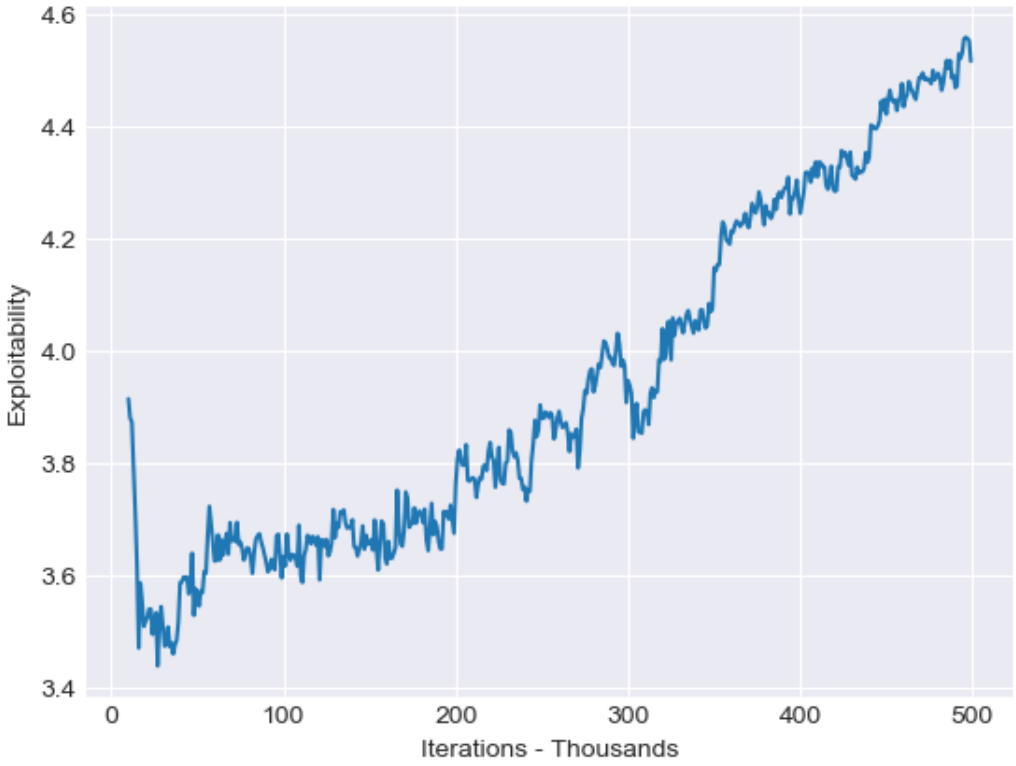

Figure 4.5: UCT exploitability - self play - 500000 Iterations

4.3.4 Analysis

The results gleaned from this experiment were unexpected to a degree. Heinrich had shown exploitability reaching lows of .8 and diverging to 1.5 in a similar experiment. Although a similar trend was shown here our exploitability values were significantly higher. This could potentially be explained by differences in the algorithm used to calculate exploitability or the implementation of UCT itself.

On further inspection of the search trees generated, along with the best response tree (see chapter 3) there appeared to be an over-fitting to the opponent's strategy over time. Notably, both instances of the agent were playing more conservatively than expected. This meant that the best-response player could regularly force our agent to fold in situations where folding against a

less conservative player would be illogical. In order to conceptualize this phenomenon, the following example is given. Let's say player one is very conservative and they decide to only raise in the second round of the game when they have a pair of aces. This means that over time player two will only receive highly negative rewards for the states in which player one has raised in the second round. In fact, in this case, it is more beneficial for player two to simply fold if player one raises in the second round of the game. When a new player is introduced to the system, however, they can take advantage of this behavior by simply raising more frequently in the second round of the game. This type of strategic feedback loop is characteristic of deterministic strategies due to the fact that breaking such a loop becomes less and less likely the closer we get to purely greedy action selection.

4.4 Experiment 3 - Smooth UCT

The final experiment conducted during our empirical studies utilised both self-play and average strategy sampling. Ie smooth UCT was used.

4.4.1 Objective

The objective of this experiment was to implement Smooth UCT as outlined in(Heinrich 2017). Again a significant improvement in exploitability was expected at this stage due to the fact that stochastic strategies were now in use in the training phase. Heinrich showed smooth UCT to converge to an approximate Nash's equilibrium for Leduc Hold'em. As such the primary goal for this experiment is to replicate these results as closely as possible. This goal was set with the caveat that, as mentioned in section 3.2.2 only deterministic strategies could be evaluated for exploitability at this stage of development. As such the exploitability values produced may be higher than expected.

4.4.2 Experimental Parameters

Note that in the case of smooth UCT the d parameter has been modified, with Heinrich's original value being .002. It was found through experimentation that Heinrich's stated value did not provide a significant improvement compared to experiment 2's results. However, when increased to .1 we saw our best exploitability results.

Below are listed the key parameters of the experiment:

- **Iterations** - 1,000,000
- **Repetitions** - 10
- c **- exploration constant** - 18
- γ - .01
- η_0 - .9
- d - .1

4.4.3 Results

For this experiment exploitability was once again the sole metric used. In this case, the algorithm was run for 1,000,000 iterations and 10 repetitions with the results averaged. The results of the experiment are shown in figure 4.6.

Here we see a significant improvement compared to the last experiment in which deterministic strategies were used. There is no longer a tendency for exploitability to diverge as there it had in experiment 2 with early exploitability values of 3.8 converging towards 2.2 and stabilising. However, these values are still less favourable than those demonstrated by Heinrich who achieved exploitability of roughly .5 after 1,000,000 iterations.

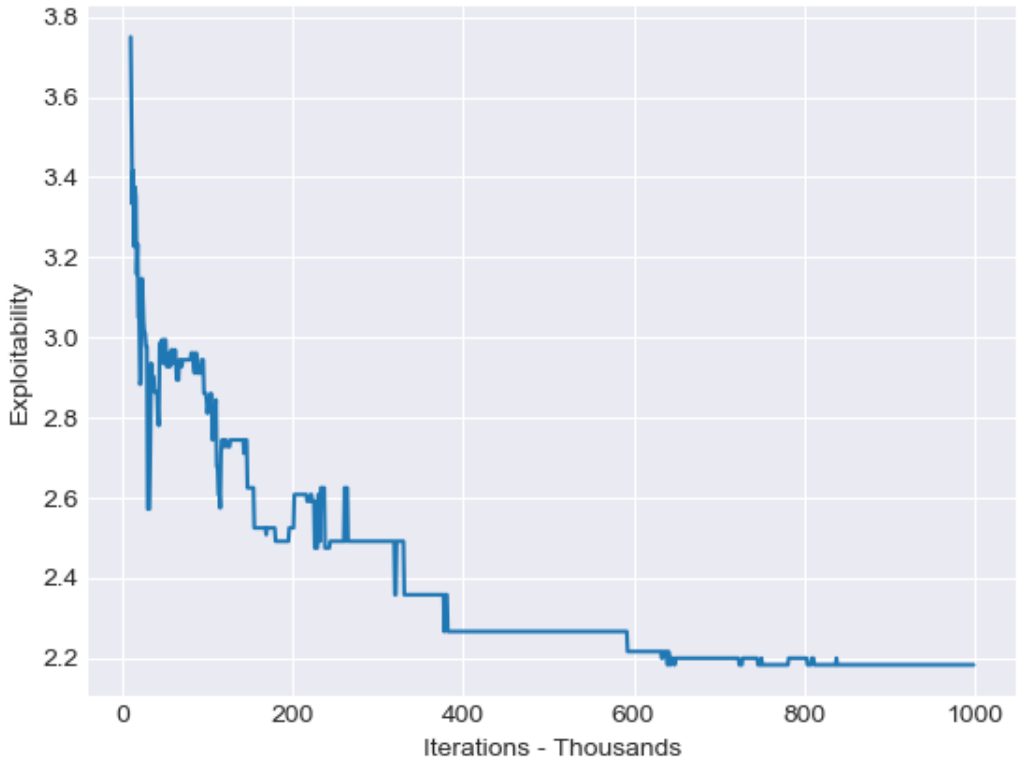

Figure 4.6: Smooth UCT exploitability - self play - 1000000 Iterations

4.4.4 Analysis

The relative success of this experiment compared to the others demonstrates the value of utilising stochastic strategies in these scenarios. The over-fitting effect mentioned in experiment 2 was no longer an issue. This is likely due to the fact that when the average strategy is being sampled there is a chance to break out of such a feedback loop as all possible actions will be sampled over time.

Although Heinrich's benchmark was not met, there are a number of extenuating factors that must be considered in order to contextualize these results. Firstly, the starting exploitability value is significantly higher than Heinrich's (3.8 vs 2). This may indicate that the general methodology for exploitability computation differed across these two implementations. Second, due to the

scope of the project extensive parameter tuning was not possible. It's most likely that these results could be significantly improved if the optimal parameter combination was found for this implementation. Third, as mentioned, only exploitability calculation for deterministic strategies was performed. If this had been implemented for stochastic strategies then the exploitability values would likely have been lower due to the fact that stochastic strategies fare far more favourably in imperfect information games(Heinrich & Silver 2016). Finally, the computational resources available for this project were limited and as we could not fully replicate Heinrich's experiments. It would have taken a number of days to complete 500,000,000 iterations of the algorithm on the available hardware.

Chapter 5

Conclusions

In this chapter, there will be a review of the work done throughout the course of this project. A summary of the findings will be given along with a reflection on how the project was conducted. Finally, a brief discussion on future work will be given.

5.1 Summary

The initial goal of this final year project was to explore the possibility of applying reinforcement learning techniques to imperfect information games. The initial focus of the project was on the game of Texas hold'em with research being done into different RL methods that could be used in order to create an agent for the game. It was decided that Monte Carlo Tree Search would be the method used and there would be an attempt to recreate the results demonstrated in(Heinrich 2017).

When it came to creating a prototype implementation the poker variation used was a scaled-down version of Texas hold'em called Leduc Hold'em. A number of experiments were conducted in which a variety of training methods were used for the agent. From the first to last experiment we saw a significant improvement in performance of the agent with the results trending in the same manner as Heinrich had reported. Although our results trended as

expected we did not achieve the same numeric values as Heinrich, with our agent's exploitability(2.2) being higher than his (.5) after 1,000,000 iterations of the smooth UCT algorithm.

Despite the fact that exact replication of Heinrich's results was not achieved we can think of the improving trend of our results to be a sign of success. This is especially true when we consider the fact that Heinrich did not provide an algorithm for exploitability calculation and thus we cannot be sure that our metrics were exactly the same.

When the Smooth UCT algorithm had been implemented, a prototype game was created that allowed users to interact with the trained agent.

5.2 Reflections

Although the goals set out for the project were not fully achieved, the project can still be thought of as a success. We provided the first known open source implementation of Heinrich's smooth UCT algorithm and documented the implementation process, making it easier for future researches implement and utilise this algorithm. Due to the fact that this algorithm has been shown to be applicable to limit Texas hold'em(Heinrich & Silver 2015) we see this as a significant achievement.

If I were to attempt this project again with my current knowledge more time would be spent on the implementation process of the algorithm. A significant portion of the project was spent gaining the requisite background knowledge in order to be able to attempt an implementation. If more time was available for this implementation then through the use of parameter tuning, the further development of the exploitability calculation algorithm and additional training time would almost certainly result in vastly improved results.

I feel that the use of the Python programming language for this project was a good choice, its concise syntax, extensive set of support libraries and easy to use data structures made the implementation process fast and effi-

cient. The same can be said for Qt Designer and PyQt when implementing the prototype game. These libraries aided in facilitating the fast and easy development of a user interface despite the fact that I had no previous experience in these technologies.

5.3 Future Work

The next large task for this project would be to tackle a more complex version of the game. In this second iteration of the project limit Texas hold'em would be tackled with the end goal of recreating the results shown by Heinrich in his PhD thesis(Heinrich 2017). In this case, we would measure win rate against other texas hold'em agents in order to evaluate the success or failure of our product.

The final step of this project would be to attempt to extend the same methods to no-limit Texas hold'em. This game has a much larger state space(circa. 10^{164}) than limit texas hold'em(circa. 10^{17}). This final approach would likely require the introduction of neural networks that would provide accurate generalisation between states in order to reduce the complexity of the problem.

Bibliography

Dahl, F. A. (2001), A reinforcement learning algorithm applied to simplified two-player texas holdem poker, *in* 'European Conference on Machine Learning', Springer, pp. 85–96.

Desjardins, C. & Chaib-Draa, B. (2011), 'Cooperative adaptive cruise control: A reinforcement learning approach', *IEEE Transactions on intelligent transportation systems* **12**(4), 1248–1260.

Heinrich, J. (2017), Reinforcement Learning from Self-Play in Imperfect-Information Games, PhD thesis, UCL (University College London).

Heinrich, J., Lanctot, M. & Silver, D. (2015), Fictitious self-play in extensive-form games, *in* 'International Conference on Machine Learning', pp. 805–813.

Heinrich, J. & Silver, D. (2015), Smooth uct search in computer poker, *in* 'Twenty-Fourth International Joint Conference on Artificial Intelligence'.

Heinrich, J. & Silver, D. (2016), 'Deep reinforcement learning from self-play in imperfect-information games', *arXiv preprint arXiv:1603.01121* .

Jackson, E. (2013), Slumbot nl: Solving large games with counterfactual regret minimization using sampling and distributed processing, *in* 'AAAI Workshop on Computer Poker and Incomplete Information'.

Johanson, M. (2013), 'Measuring the size of large no-limit poker games', *arXiv preprint arXiv:1302.7008* .

Johanson, M., Waugh, K., Bowling, M. & Zinkevich, M. (2011), Accelerating best response calculation in large extensive games, *in* 'Twenty-Second International Joint Conference on Artificial Intelligence'.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996), 'Reinforcement learning: A survey', *Journal of artificial intelligence research* **4**, 237–285.

Koller, D., Megiddo, N. & Von Stengel, B. (1996), 'Efficient computation of equilibria for extensive two-person games', *Games and economic behavior* **14**(2), 247–259.

Kuhn, H. (1953), 'Extensive games and the problem of information', *InH. KuhnandA. Tucker, editors, Contributions to the Theory of Games* pp. 193–216.

LazyProgrammerInc. (2018), 'Artificial intelligence: Reinforcement learning in python'.
URL: *https://www.udemy.com/artificial-intelligence-reinforcement-learning-in-python/*

Lee, M. (2005), 'Monte carlo control'.
URL: *http://www.incompleteideas.net/book/ebook/node53.html*

Leslie, D. S. & Collins, E. J. (2006), 'Generalised weakened fictitious play', *Games and Economic Behavior* **56**(2), 285–298.

Lim, J. & Yoo, S. (2016), Field report: Applying monte carlo tree search for program synthesis, *in* 'International Symposium on Search Based Software Engineering', Springer, pp. 304–310.

Mitchell, T. M. et al. (1997), 'Machine learning. 1997', *Burr Ridge, IL: McGraw Hill* **45**(37), 870–877.

Myerson, R. B. (2013), *Game theory*, Harvard university press.

Neller, T. W. & Lanctot, M. (2013), An introduction to counterfactual regret minimization, *in* 'Proceedings of Model AI Assignments, The Fourth Symposium on Educational Advances in Artificial Intelligence (EAAI-2013)'.

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

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. et al. (2016), 'Mastering the game of go with deep neural networks and tree search', *nature* **529**(7587), 484.

Silver, D. & Veness, J. (2010), Monte-carlo planning in large pomdps, *in* 'Advances in neural information processing systems', pp. 2164–2172.

Southey, F., Bowling, M. P., Larson, B., Piccione, C., Burch, N., Billings, D. & Rayner, C. (2012), 'Bayes' bluff: Opponent modelling in poker', *arXiv preprint arXiv:1207.1411* .

Sutton, R. S., Barto, A. G., Bach, F. et al. (1998), *Reinforcement learning: An introduction*, MIT press.

Tesauro, G. (1995), Td-gammon: A self-teaching backgammon program, *in* 'Applications of Neural Networks', Springer, pp. 267–285.

Vermorel, J. & Mohri, M. (2005), Multi-armed bandit algorithms and empirical evaluation, *in* 'European conference on machine learning', Springer, pp. 437–448.

Von Stengel, B. (1996), 'Efficient computation of behavior strategies', *Games and Economic Behavior* **14**(2), 220–246.

Watkins, C. J. C. H. (1989), Learning from delayed rewards, PhD thesis, King's College, Cambridge.

Weng, L. (2018*a*), 'A (long) peek into reinforcement learning'.
URL: *https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html*

Weng, L. (2018*b*), 'The multi-armed bandit problem and its solutions'.
URL: *https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html*

Zinkevich, M., Johanson, M., Bowling, M. & Piccione, C. (2008), Regret minimization in games with incomplete information, *in* 'Advances in neural information processing systems', pp. 1729–1736.