**David Moody**

# Monte Carlo Tree Search in Texas Hold 'em Poker

Computer Science Tripos, Part II

Christ's College

May 19, 2011

# Proforma

| | |
|---|---|
| Name: | **David Moody** |
| College: | **Christ's College** |
| Project Title: | **Monte Carlo Tree Search in Texas Hold 'em Poker** |
| Examination: | **Computer Science Tripos, Part II, 2010-2011** |
| Word Count: | **Approximately 12,000 words** |
| Project Originator: | Dr Sean Holden/David Moody |
| Supervisor: | Dr Sean Holden |

## Original Aims of the Project

To build a computer poker player using the Monte Carlo Tree Search algorithm. The program should also implement an opponent model and have a variety of different strategies to choose from. It should be able to consistently beat an existing simple opponent, SimpleBot.

## Work Completed

I have successfully implemented the program as originally intended. It can play as well as I had expected and can consistently beat SimpleBot. All work completed is described in this dissertation.

## Special Difficulties

None.

# Declaration

I, David Moody of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 My Project

My project was to build a program to play Texas Hold 'em Poker. The program uses the Monte Carlo Tree Search algorithm for searching large trees. It also incorporates an opponent model to assist in the presence of hidden information. I have successfully implemented the program as originally intended. It can play to an acceptable level and can consistently beat a simple opponent I chose in my original plan, SimpleBot. The program and this dissertation successfully complete all of my original success criteria. However, I did not have time to complete any of my planned extensions.

## 1.2 The Game of Poker

Poker is a game of imperfect information, where the other players know different things and may try to deliberately mislead you. There is also an element of chance in what cards will come up next. In addition, there can be as many as 10 players in a game which can result in an enormous number of possible game states. These things make it particularly difficult for computers to play poker.

Despite this, there have been many successful attempts to build competent computer poker players (aka poker bots). Here is a brief overview of just a handful of them:

- Poki, 1999: A full-ring limit poker bot, its main problem was that it could not adapt its strategy fast enough to prevent its own exploitation [5].
- PsOpti, 2002: A heads up limit poker bot, built using a game theoretic approach [7].
- Vexbot, 2003: A heads up limit poker bot which uses opponent modelling and the expectimax algorithm to adapt to its opponents [17].
- Polaris, 2007: Another heads up limit poker bot which contains a number of different strategies and chooses between them during a match [11].

## 1.3 Monte Carlo Tree Search

Monte Carlo Tree Search is a best first search technique used to search very large trees. It uses stochastic simulations to help decide which action to take. It tries to focus its simulations onto the branches which it thinks will be most relevant, assuming all players play rationally. Every time it simulates a game, it backpropagates the result back through the tree so that it can make better decisions about which game to simulate next.

MCTS has been successfully been applied to Go [4], Backgammon [12], Limit and No-Limit Texas Hold 'em [15] [6] as well as several other games.

In this project I will be using MCTS with a variety of different strategies. In the evaluation, I will compare the effectiveness of some of the different strategies and see how varying the thinking time affects performance.

## 1.4 Opponent Modelling

An opponent model can be used to predict the cards that the opponents hold and the actions they will take. There has been quite a lot of research on different opponent modelling techniques in poker including some models which can dynamically adapt to different players.

In this project, I will be using relatively simple opponent models which use classifiers created by an open source machine learning library called Weka. My approach is similar to that taken by [6].

I eventually found that opponent modelling is very important to the success of the program. Without it, the program was not able to beat SimpleBot.

## 1.5 Results

The project has been a great success. The program is able to beat the simple opponent originally proposed in the project proposal, SimpleBot. It can also be the most successful player when played against three separate instances of SimpleBot. I have also shown how varying different factors, such as the thinking time or the presence of the opponent models, affects performance.

The program and this dissertation complete all 7 of my original success criteria.

# Chapter 2

# Preparation

## 2.1 The Rules of Poker

To help you better understand this project, I will now briefly explain the rules of Limit Texas Hold 'em Poker.

There is a minimum of two players and usually no more than ten. At the start of each game, each player is dealt two cards, face down so that only they can see them. These cards are often called the hole cards.

Then there is a round of betting. In a round of betting, going clockwise through the group starting with the player to the left of the dealer, each player must choose to perform one of the following possible actions: they can fold, and drop out of the game; they can call, and place into the pot the current maximum bet; or they can raise, and increase the current maximum bet by a fixed amount. Note that calling when the current maximum bet is zero is usually called checking. The betting round continues until all players have either folded or matched the current maximum bet.

Here is a quick example betting round between three players, Alice, Bob and Charlie:

| Player | Action | Amount in pot from player before/after | Total amount in pot before/after | Current maximum bet |
|---|---|---|---|---|
| Alice | Check | \$0 / \$0 | \$0 / \$0 | \$0 |
| Bob | Raise | \$0 / \$1 | \$0 / \$1 | \$1 |
| Charlie | Call | \$0 / \$1 | \$1 / \$2 | \$1 |
| Alice | Raise | \$0 / \$2 | \$2 / \$4 | \$2 |
| Bob | Call | \$1 / \$2 | \$4 / \$5 | \$2 |
| Charlie | Fold | \$1 / \$1 | \$5 / \$5 | \$2 |

After this round of betting, if at least two players are still in the game, three cards are dealt face up onto the table. There is another round of betting, another card on the table, another round of betting, a final card on the table and then a final round of betting. The four betting rounds are usually given the following names: preflop, flop, turn and river.

If all but one of the players folds then the remaining player takes all of the money in the pot. If the final betting round has ended and there are still two or more active players then there is a showdown; all active players show their cards and the player who can make the best poker hand wins the pot. It is also possible for multiple players to be able to make the same best poker hand. In this case, they will each take an equal share of the pot.

The best poker hand of each player is made up of the best combination of their two hole cards and the five cards on the table. The exact ranking of a poker hand is quite complicated and not really necessary to know for this project so I will not describe it in any more detail. All you need to know is that there are relatively fast algorithms for finding a numerical rank of a poker hand such that any better hand will have a higher rank, any worse hand will have a lower rank and any equal hand will have the same rank.

In Limit Texas Hold ‘em, the small bet is the amount that one is allowed to increase the current maximum bet by (in the first two betting rounds). In the above example the small bet is \$1. In the last two betting rounds, the amount that one is allowed to increase the current maximum bet by is usually doubled (to \$2 in this example). To avoid any confusion, throughout this project the small bet will always be equal to \$1. Also there is a limit to how large the current maximum bet can be. The limit is usually four times the raise increment (so \$4 in the first two rounds of betting and \$8 in the last two). This puts a limit on length of the game.

In the first round of betting (preflop) there is an additional rule. The player to the left of the dealer must place into the pot the small blind and the player to the left of that player must place into the pot the big blind. These blinds are similar to regular bets except they are not optional. The small blind and big blind are usually equal to one half of the small bet and one small bet respectively. The main purpose of the blinds is to get the betting going and prevent the players from all checking in the first round. Also, if no one else raises in the round then the player who put in the big blind is given the opportunity to raise.

There are a lot of complex rules regarding what to do when a player has run out of money but is still in the game. However these situations are very rare so I will not consider them.

There are many other varieties of poker including one where players can raise any amount they want. The reason I chose Limit Texas Hold 'em is because Texas Hold 'em is the most popular variety (and is the version I play) and because designing a No-Limit bot would have been too complicated. Also note that throughout the project, I will be referring to cash games as opposed to tournament games.

## 2.2 Poker Academy Pro

Poker Academy Pro [10] is a commercial program[1] which provides a variety of useful and fun features for playing and analysing poker games. The two features that are relevant for this project are: the ability to play user created custom poker bots against a variety of existing poker bots; and the ability to record and analyse various statistics about the games played.

The Meerkat API [9] provides the ability to create a plug-in bot which can play in the Poker Academy program; all that is required is to create a Java class that implements a given interface.

I chose to use Poker Academy because of the graphical user interface, because of its statistics tracking capabilities and because it was highly recommended. There were two alternatives. The first was to write the code for simulating the games myself. This would have been very difficult and would have taken too long. The second possibility would have been to use opentestbed [16]. Opentestbed is an

[1] It costs $85 from the Poker Academy website.

open source implementation of the Meerkat API for running poker games. In retrospect, I think it would have been better to use opentestbed. It was designed specifically for the purpose of testing poker bots and may have been easier to use.


Figure 2.1: A screenshot of Poker Academy Pro.

## 2.3 SimpleBot

SimpleBot is an example poker bot provided with the Meerkat API. The important points to note about SimpleBot are:

- It only considers its own hole cards, the table cards, the amount in the pot and the number of active players.
- It ignores the past actions of all players.
- It is non-deterministic and has the ability to bluff.

SimpleBot's playing ability is not comparable to that of an experienced human's, however it is good enough to provide a challenge for another poker bot.

I chose to use SimpleBot as the opponent to test and evaluate my program against throughout the project. I did this because I needed an opponent to test against and I didn't want to be in competition with some of the really strong poker bots that have been in development for years. Thus, I settled on SimpleBot, a good enough opponent to provide a challenge but not so good that I wouldn't stand a chance.

## 2.4 The Game Tree

A game tree is a directed graph where each node represents a possible state in the game. An edge between nodes represents an action that transforms the parent node into the child node. The root node is the initial position and the leaf nodes represent positions in which the game has ended.

For games which have perfect information and no random elements, it is possible to search the tree using the minimax algorithm. There are extensions to the minimax algorithm which allow it to deal with random elements (expectiminimax) and larger trees (alpha-beta pruning or limiting depth search combined with an evaluation function). However, there are a few problems with these methods and they are not always suitable for poker.

In the poker game tree, there are several different types of node:

- Choice node, this is where the next action will be performed by the bot. It can choose to raise, call or fold and the children of this node will reflect the choice that was made.
- Opponent node, this is where the next action will be made by one of the opponents. They can raise, call or fold.
- Chance node, this is where the current betting round has ended and one or more new cards are about to be dealt onto the table. This node can have either 46 or 47 children in the flop or turn stages, where one card is about to be dealt, or 50*49*48=117600 children in the preflop stage, where three cards are about to be dealt.

- Leaf node, this is where the game has ended. This node type has no children. There are three possible subtypes of this node:
  - All opponents folded, this is where the bot is the only remaining player in the game; it wins by default.
  - Bot folded, this is where the bot has folded but other players are still in the game. The game might not necessarily have ended, however since the bot is no longer playing, there is no point in continuing.
  - Showdown, this is where the bot and one or more other opponents are still playing and the final betting round has ended. In this case, it is not possible to say who wins because there is no way to tell what cards the opponents have. The result of the game has to be estimated from the strength of the best poker hand that the bot can make and the behaviour of the opponents up to this point.

## 2.5 Monte Carlo Tree Search

The Monte Carlo Tree Search algorithm gradually builds up a subsection of the full game tree by starting at the root node and adding a new node in each iteration. The new nodes are not added at random though, the algorithm tries to add more nodes to the tree in places which it thinks are more likely to be relevant.

For example, if an opponent has been raising all game then that opponent is unlikely to fold in the final betting round. Thus the MCTS algorithm will spend more time building the subtrees where the opponent does not fold than the subtrees where the opponent does.

One of the advantages of the MCTS algorithm is that it does not need to search the tree exhaustively and can be halted at any point. This is extremely useful because the full game tree can easily become very large and searching the entire tree could take more time than is available.

The actual body of the algorithm is fairly simple. It consists of four main stages which are repeated until the program runs out of thinking time. The four stages are:

1. Selection: Here, starting at the root of the tree, the algorithm selects the next node that it wishes to look at and repeats until it has reached a leaf node of the stored tree. The selection stage focuses the algorithm onto the paths which appear to be the most relevant.

2. Expansion: One (or more) children are then added to the node selected in the last stage.

3. Simulation: The game is simulated (to completion) from the state at the newly added node to attain an estimate for the expected value of the node.

4. Backpropagation: Once a simulation result has been calculated, the algorithm goes back through the path that had been taken and updates the visit counters and expected values of all of the nodes that it encounters.

Then, the actual action that is to be taken needs to be selected. It is usually the one which has the highest expected value.

For the selection, simulation and backpropagation stages, there are several different strategies that can be employed. In the implementation chapter, I will describe the strategies which I used and in the evaluation chapter I will compare their effectiveness.

## 2.6 Opponent Modelling

Opponent modelling is where one attempts to create a model to predict certain information about ones opponents. Two useful things that can be predicted are:

1. The hand rank of the best poker hand an opponent can make given the cards on the table and the opponent's previous actions.

2. The probability that an opponent will perform a given action at a particular stage given the cards on the table and the opponent's previous actions.

The first can be useful when trying to predict the result of a simulated game and the second can be useful when selecting the next node to examine (where the next action will be made by an opponent). Without an opponent model, a variety of different problems can occur. For example, if an opponent is showing signs of

weakness then it is often a good move to raise and force them to fold. However, if an opponent model is not being used then this option may be overlooked.

I found that implementing the opponent model had a major effect on the performance of the program; it was the first thing I did that enabled the program to start beating SimpleBot.

## 2.7 Weka

Weka (Waikato Environment for Knowledge Analysis) [14] is a collection of machine learning algorithms written in the Java programming language. Weka is available under the GNU General Public License.

Weka provides several features that I needed for this project. The most important is the ability to create classifiers for use in the opponent models. It also provides a useful GUI to help in analysing the training data and creating the classifiers. Weka has been used for creating opponent models in similar poker bots [6] so I was confident that it would work here.

The alternatives to using Weka would have been to create my own machine learning algorithms, which I did not have time to do, or to use a different software package, for example Matlab. In the end I chose to use Weka because it runs on Java, which is also used by Poker Academy Pro, and because it looked easy to use.

Note that even though Weka produced the classifiers, I still had to write a significant amount of code to integrate them into the program.

Unfortunately, because Poker Academy runs with Java 1.5, I was not able to use the most recent version of Weka and had to use an older version from 2004 (version 2.3.3). I doubt this had much of a negative effect on the project as version 2.3.3 had all the required features.

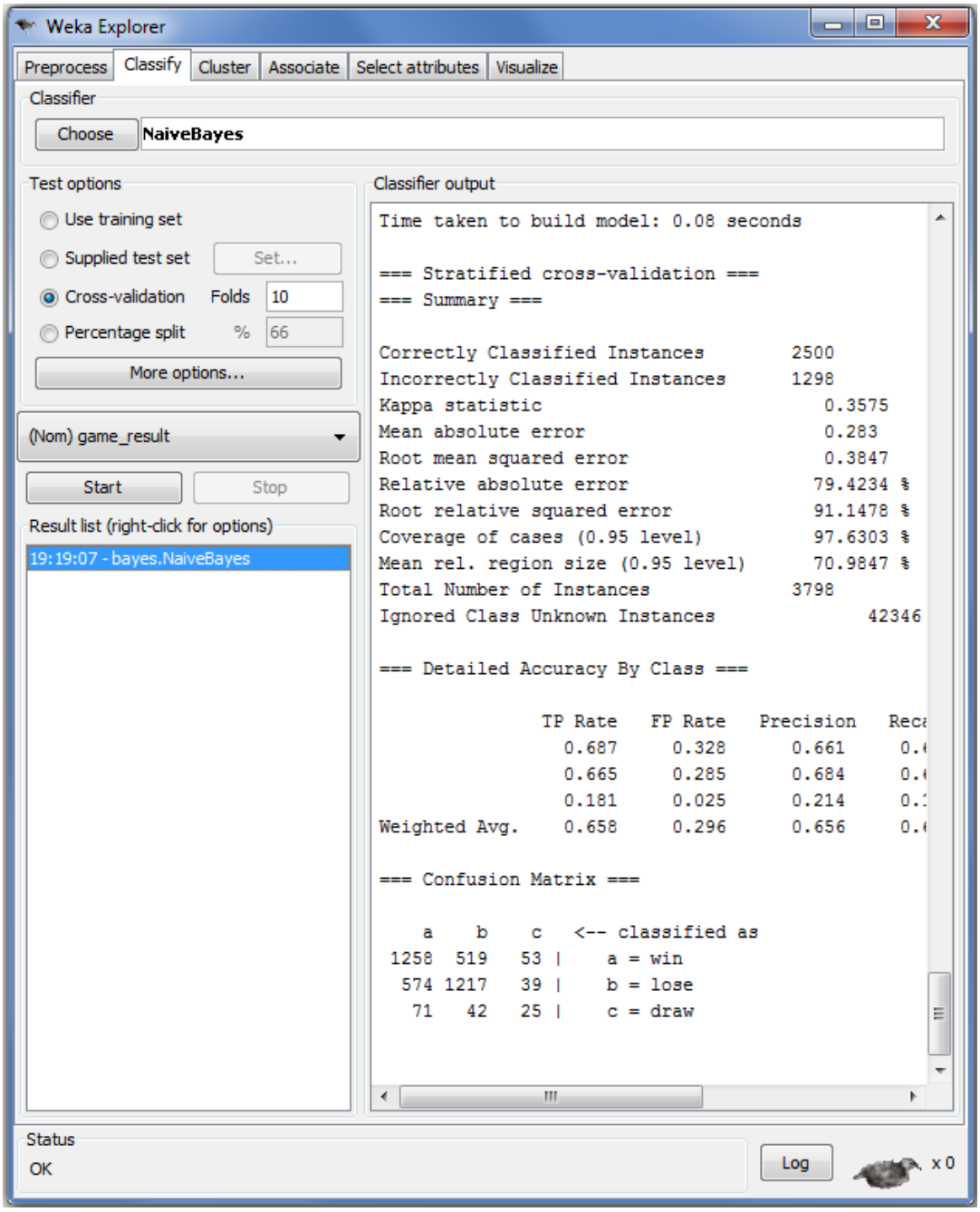


Figure 2.2: A screenshot of the Weka Explorer.

## 2.8 Training Data

In order for Weka to create a classifier, it needs to be provided with training data. In this case, training data takes the form of records of played games (called hand

histories). I had originally planned to get hand histories from a source on the internet [1]. I eventually realised that I could get hand histories more easily by playing SimpleBot against another instance of itself inside Poker Academy Pro. Early on in the project, I set up a match of SimpleBot Vs. SimpleBot and let it run for 50,000 games. I used that as the training data for all of the classifiers I created.

There is also a problem of converting the hand histories into a format that Weka can recognise. Hand histories exported from Poker Academy (or from the web) come as human readable plain text descriptions of the games. Before I could use them as training data for Weka, I first had to convert them into Attribute Relation File Format (ARFF). I wrote my own tools to do this as there are no freely available ones.

## 2.9 The Plan

In my original plan, I proposed to complete the work in the following order:

1. Create the basic classes needed, implement the game logic and integrate with Poker Academy Pro.
2. Implement the framework for the MCTS algorithm and create basic strategies.
3. Implement more complex strategies for the MCTS algorithm.
4. Collect and consolidate hand histories.
5. Use Weka to create opponent models and integrate them into the main program.
6. Evaluation and testing.

I chose to do it in this order because it allowed me to produce a working poker bot after the first term. It would have also allowed me to complete the project without the opponent model if the implementation of the MCTS algorithm had taken too long.

Throughout the project, I was using Mercurial for version control and Dropbox to backup the repository. I had also intended to write good documentation for all of the classes in the project[2].

[2]However, due to time constraints, I was not always able to do this.

# Chapter 3

# Implementation

## 3.1 Integration With Poker Academy Pro

One of the first challenges I faced was getting my program to correctly interface with Poker Academy Pro. In this section I will briefly discuss how I did this and give a quick description of some of the useful features of the Meerkat API.

Recall that Poker Academy allows users to create their own plug-in bots. To do this, the Meerkat API is required. The Meerkat API contains an interface called Player which all plug-in bots must implement. My bot, which is called MCTSBot, implements the Player interface.

I will not describe the API in great detail but there are three important methods in the Player interface: getAction, this is called every time Poker Academy wants to know what action the bot wants to perform and it is where the majority of the work is done; init, this is called when the bot is first loaded, it allows the bot to initialise itself with some given preferences; and holeCards, which is called once per game to tell the bot what its hole cards are. There are also eight other methods which are used to notify the bot whenever any action occurs in the game. I implemented most of these methods in order to keep a record of the past actions of the other players.

The Meerkat API also provides a few other useful classes: the Card class, which provides a basic representation of a card; the Deck class which provides an easy way to extract a random Card; the Hand class which is a collection of 5-7 Cards;

and the HandEvaluator class which is used for determining the numerical hand rank of a 5-7 Card Hand. There are also several other classes which I did not use.

## 3.2 The Game Logic

In this section, I will briefly talk about how I implemented the game logic. The game logic is required for determining how performing different actions will change the state of the game. It is needed in the expansion stage of the MCTS algorithm for creating new nodes and also in the simulation stage.

The GameState class contains all of the information about the current state of the game, including the cards on the table, the amount in the pot, a list of the players in the game, etc. The class also has a total of 23 methods for accessing this information.

The Player class contains all of the information about one player in the game at a particular moment in time. It stores the amount of money the player has put into the pot (in total and in the current round) and a list of all of the previous actions that the player has made in the current game. The previous actions are stored because they are needed by the opponent models.

Both GameState and Player are immutable and cannot be altered once they have been created. The reason for this is that every node in the tree needs to have its own GameState object and if the class is immutable then there is no possibility of inadvertently changing a previous GameState further on down the tree.

There are two ways to create a new GameState object, the first is to use the static initialise method and provide it with a GameInfo object[1], the second is to call one of several possible methods in an existing GameState object. These methods represent the different ways in which the game state can be advanced. For example, there are methods for making the next player to act perform a specified action, dealing a new card onto the table and advancing to the next stage of the game. When one of these methods is called, it first clones itself and

[1]The GameInfo class is from the Meerkat API, it is similar to the GameState class in that it stores information about the current state of the game. However, it does not contain information about the past moves or provide any way to simulate the effects of performing particular actions. A new GameInfo object is given to MCTSBot at the start of each game.

then updates the information that needs to be changed in the new copy. It only takes a shallow copy of itself so it can reuse the Player objects (unless the Player object is question needs to be changed). This saves memory when the program is being run.

I will not go into any more detail about how I implemented these methods because it is rather tedious and relies heavily on the specifics of the rules of poker.

## 3.3 Monte Carlo Tree Search

In this section, I will describe how I implemented the framework for the Monte Carlo Tree Search algorithm. In subsequent sections, I will discuss the different strategies I used.

Because I knew that I would be implementing alternative strategies later on, I decided to use a design pattern called the strategy pattern. This involves creating one interface (for each different type of strategy) and then writing multiple classes which implement it. Whenever one of the strategies is needed, its interface is used. This provides the ability to easily switch to a different strategy at compile-time (or run-time if needed).

I created interfaces for the selection strategy, the simulation strategy, the back-propagation strategy, the action selection strategy and both of the opponent models. I also created a class called Config, it contains one object for each of those interfaces as well as a get method for each one. Whenever anything needs to use one of the strategies, it gets it from the Config class. I created the Config class because it allowed me to make all of my strategy choices in one class, rather than having to search through multiple different files.

The MCTS algorithm also needs a game tree to work with. I created one abstract class called Node. It contains the information required by every node type, that is: a list of its children, a reference to its parent, a GameState, a visit count, an estimated expected value and a variety of other useful fields and methods used by the MCTS algorithm. I also created one subclass for each type of node described in the game tree section in the preparation (including the different types of leaf node). Some of these subclasses override methods in the Node class where appropriate. I will go into more detail in later sections.

The MCTS algorithm is run once whenever MCTSBot is called upon to perform an action. Before the main body of the algorithm can start, the root node must first be created. I wrote a class called RootNode which extends ChoiceNode. It has a static method to create an instance of itself from the data stored in a GameInfo object.

Once the root node has been created, it is passed to a private method called performIterations. This method repeatedly calls the iterate method on the root node until its allotted time has run out. The iterate method performs one complete iteration of the MCTS algorithm. First it calls the selectRecursively method of the root node, this repeatedly calls select on the nodes down a path on the tree until it reaches a leaf node of the stored tree. It then calls the generateChildren method on the selected child node and then select again to select one of the newly generated children. It calls simulate on this new child node to get an estimate of the expected value of the child. Once it has this, it calls backpropagate to update all of the nodes in the path taken.

Once the given thinking time has elapsed, the performIterations method finishes and the current action selection strategy is used to select the actual action to take. I used a simple action selection strategy which always selects the action with the highest expected value. The selected action is then converted from one of my own action classes into an instance of the Action class from the Meerkat API.

The basic MCTS algorithm is relatively simple and it is the different strategies which add more depth. In the next four sections I will describe the stages in more detail and talk about the strategies I implemented.

## 3.4 Selection

The selection strategy's select method is used to select the next node to examine. Repeated calls of the select method are chained together to traverse from the root node to a leaf node in the stored tree[2]. The main purpose of the selection strategy is to focus the algorithm onto the paths that are most relevant and most likely to actually occur. I will now describe the different strategies that I used.

[2]Note that I am referring to a leaf node in the partially built subtree stored in memory which is not necessarily a leaf node of the full game tree.

### 3.4.1 Random Selection Strategy

The first selection strategy I implemented was the random selection strategy. As the name suggests, this strategy will always randomly select one of the child nodes (where each child node has an equal probability of being selected).

The main purpose of this strategy is to act as a fall-back to be used by the more complex strategies whenever they encounter a node type which they are unable to deal with.

### 3.4.2 UCT Selection Strategy

When selecting for a ChoiceNode, it is generally best to select the options which have the highest expected value. This is because the goal is to maximize the pay-off. However there is always the possibility that, due to the randomness of the simulation strategy, a good path will be estimated to have a lower expected value than it should have. Because of this it is necessary to explore all possible options, to some extent, even if they initially appear to have low expected values.

There is a complex balance between exploitation and exploration. If too much time is spent exploiting then there is a risk of discarding better options just because they were not examined closely enough. If too much time is spent exploring then there will be less time to look at the most promising options.

I used a method called **U**pper **C**onfidence Bound applied to **T**rees (UCT). It is supposed to provide a good balance between exploitation and exploration and it has been used in many similar poker bots [6][8][15][18].

The formula is:

$$\text{selected child index} = \text{argmax}_i \left( v_i + C\sqrt{\frac{\ln \sum_j n_j}{n_i}} \right)$$

where $v_i$ is the current estimate of the expected value of the $i_{th}$ node, $n_i$ is the current visit count of the $i_{th}$ node and C is a constant to be determined experimentally.

This requires all nodes to have been selected at least once before. If the UCT strategy is called when there are still some nodes with visit counts equal to zero, it will randomly select one of them instead.

To implement this, I created a method which simply calculates the value in the brackets for each node and then returns the node with the highest value. Also note that the UCT strategy delegates to the random strategy for all node types except ChoiceNode.

To choose the value for the constant $C$, I added a small amount of code to the select method of the UCT strategy. This code kept track of the number of times that a node was returned where that node already had the maximum expected value out of its siblings and the number of times where a node was returned where it did not already have the maximum expected value. These two tallies represent the proportions of time that the strategy engaged in exploitation and exploration respectively. Several other papers mentioning UCT have observed that spending somewhere around 5% of the time on exploration can give good results. By a process of trial and error, I eventually settled on a value of 10 for the constant $C$.

### 3.4.3 UCTVar Selection Strategy

An extension to the UCT strategy was proposed in a recent paper [6]. The general idea is to select the child which has the highest:

$$v_i + C\sigma_{v_i}$$

where $v_i$ is the current estimate of the expected value, $C$ is a constant and $\sigma_{v_i}$ is the standard error on $v_i$.

Here, the first term is the exploitation term and the second is the exploration term. This way, the strategy takes into account the uncertainty based on the actually observed samples of the child node.

There are several differences between my program and theirs so I had to slightly adapt their idea. The formula I finally used is as follows:

$$\text{selected child index} = \text{argmax}_i \left( v_i + C_1 \sqrt{\frac{\ln \sum_j n_j}{n_i}} + C_2 \sigma_{v_i} \right)$$

Note that this is the same as before except that it includes an additional term, $C_2\sigma_{v_i}$. This term is there to increase the likelihood that a child will be selected when there is a high level of uncertainty in the estimate of its expected value. It is supposed to help reduce the possibility of a good option being overlooked. I found that setting $C_1$ to 10 and $C_2$ to 0.1 worked well.

The calculation of $\sigma_{v_i}$ is not trivial and depends on the backpropagation strategy. I will discuss it in the backpropagation section.

From preliminary testing, it appeared that the UCTVar strategy was slightly more effective than the UCT strategy. I will test them more thoroughly in the evaluation chapter.

### 3.4.4 UCTVar + Opponent Model Selection Strategy

One of the problems with the previous strategies is that they cannot deal with OpponentNodes. I used the next move opponent model (which I will discus in a later section) to predict the probabilities of the opponent performing each of the available actions. These probabilities are cached in the node. Then, every time the select method is called on an OpponentNode, it randomly selects one of the children according to the stored probability distribution.

This combination of the random strategy for ChanceNodes, the UCTVar strategy for ChoiceNodes and the opponent model for OpponentNodes was quite effective and it is what I used throughout the rest of the project.

## 3.5 Expansion

The expansion stage is probably the simplest stage as there is little potential for multiple strategies. All of the work is done in one abstract method called generateChildren. The generateChildren method is implemented separately in each subclass of Node. I will now explain how I implemented it in each of the Node subclasses and how I solved the problem of having too many children in ChanceNode.

### 3.5.1 ChoiceNode and OpponentNode

Both ChoiceNodes and OpponentNodes have three possible children. One for raise, one for call and one for fold. The exact node type and GameState of the children depend on the GameState of the parent node. Because of the similarities

between ChoiceNode and OpponentNode, I decided to create a superclass for them to share.

The abstract class PlayerNode extends Node and implements the generateChildren method. The generateChildren method calls three abstract methods: createRaiseNode, createCallNode and createFoldNode. Both ChoiceNode and OpponentNode implement these methods slightly differently. For example, the createFoldNode method in ChoiceNode will always create a BotFoldedNode where as the createFoldNode method in OpponentNode can either create an AllOpponentsFoldedNode or one of the other node types if there are still other players in the game. Separating the node creation methods like this is also useful to the simulation strategy.

There are also two special situations in which there should be less than three children. If a player can check then they would never chose to fold and so the fold node is not needed. Also, if the maximum current bet is already at the maximum allowed bet then no player is allowed to raise and so the raise node is not needed. In the first case, if createFoldNode is called when there is the opportunity to check then it will return the same node that createCallNode returned. Similarly for the second case.

### 3.5.2 ChanceNode

ChanceNodes can potentially have up to 50*49*48=117600 possible children in the preflop stage when three cards are about to be dealt. In later stages there can be up to 46 or 47 possible children. It would be completely impractical to attempt to generate and store that many nodes for every ChanceNode. I took a much more efficient approach which is almost functionally identical.

The only time the child nodes of a ChanceNode are ever accessed is when the select method of the ChanceNode is called. The select method in a ChanceNode will randomly select one of its children where each child node is equally likely to be selected. This is true regardless of which selection strategy is being used.

When the select method is called, it first calculates the maximum possible number of children that the node can have (either 46, 47 or 117600) and the number of children that it currently has (something between 0 and the maximum). It then generates a uniformly distributed random number between 0 (inclusive) and the

maximum (exclusive). If this number is less than the current number of children then the child at that index is returned, if not then a new child is generated and added to the list of children.

There is another method called generateChild which is used for generating one child node at a time. If only one card is to be dealt then the node keeps track of all of the cards which have already been used and makes sure not to use them again. This means that all of its children will be different.

However, if three cards are to be dealt then doing this would be inefficient. Instead, whenever a new child is generated, three random cards are used regardless of whether that combination of three cards had already been used in another node. This means that not every possible combination will be considered and some may be used more than once. It is unlikely that this would have much of an effect on the results and so I considered it a worthwhile trade off.

### 3.5.3 LeafNode

LeafNodes, by definition, have no children. For LeafNodes, the generateChildren method is left empty.

Note that when the MCTS algorithm selects a LeafNode, instead of generating a child and performing a simulation on that, it performs another simulation on the LeafNode.

## 3.6 Simulation

The simulation strategy is used to calculate an initial estimate of the expected value of a node. The most important thing about the simulation strategy is that it is fast to run. The accuracy of its predictions is not necessarily that important and it is sometimes possible for a less accurate simulation strategy to be more beneficial to the program overall. In this section, I will give a brief overview of the different simulation strategies I implemented.

### 3.6.1 Always Call Simulation Strategy

The first simulation strategy that I implemented was the always call simulation strategy. In this strategy, every player is forced to either call or check until the end of the game. This means that every player remains in the game until the end and that every player puts an equal amount into the pot.

Once the game has reached the showdown stage, the result of the game (win, lose or split the pot) is estimated and the expected value is calculated. Note that the expected value is equal to the total amount of money that the MCTSBot has after the game has ended (it is always a positive amount).

Initially, I would estimate the result of the game by dealing each opponent two random hole cards and comparing the rank of the best poker hand they could make with the rank of the best poker hand MCTSBot could make. The problem with this is that it only takes into account the cards held by MCTSBot and it ignores the opponents' actions. This can cause many different problems due to MCTSBot overestimating (or underestimating) the strength of its hand.

Once I had implemented the opponent model, I used it to predict the showdown results. It was much more effective than dealing the opponents random hole cards. I will talk about the opponent model more in the opponent model section.

One of the main advantages of the always call strategy is that it is quick to run. This is important because it must be used at least once for each node that is added to the tree.

### 3.6.2 Other Simulation Strategies

I eventually used the always call simulation strategy all of the time. However, before that I tried several other simulation strategies. In this subsection, I will describe them, why I thought they would work and why they did not work.

#### Static Distribution Simulation Strategy

The idea behind the static distribution simulation strategy was that always calling is unrealistic and so another strategy where the players can perform other actions

might produce better results.

Instead of forcing all the players to always call, this strategy plays out a game where every player's action is randomly selected from a fixed distribution. For example, in the preflop stage, there is a 65% chance that a player will fold, a 25% chance that they will call and a 10% chance that they will raise. To get suitable values, I played SimpleBot against SimpleBot in Poker Academy Pro and copied the action frequencies.

However, this did not produce the improvement I expected. At the time, I thought it was because I was not using appropriate values.

### Dynamic Distribution Simulation Strategy

I thought the static distribution simulation strategy was not working because it used unrealistic probabilities. In an attempt to improve this, I created the dynamic distribution strategy. This strategy was similar to the previous one except that, instead of using fixed probabilities, it would (dynamically) tally up the number of actions of each type that occurred in each stage while it was actually playing against SimpleBot. It would then use those tallies to calculate the probability distributions for each stage.

Perhaps predictably, this did not work any better than the static distribution strategy. This was because there was far too much variation in their predictions. For example, one time it might simulate a game where both players keep on raising right up until the final betting round and then make the opponent fold, resulting in a huge profit. Another time, it might simulate a game where both players keep on raising right up until the final betting round and then make itself fold, resulting in a huge loss.

### Selection Strategy Decorators

To try and reduce the variation in the results returned by the previous simulation strategies, I created two decorator simulation strategies. The first one takes a regular simulation strategy and an integer in its constructor. When its simulate method is called, it calculates the specified number of results using its given simulation strategy and returns the average of those results. The second one

does the same except it returns the median.

These decorators did help to reduce the variation in results. However I eventually realised that, in this case, the speed of the simulation strategy was more important than the accuracy of its predictions. After much trial and error, I switched back to the always call simulation strategy. While it does not always produce the most accurate simulation results, it does run much faster and that allows the MCTS algorithm to perform more iterations and thus produce better results overall.

## 3.7 Backpropagation

The purpose of the backpropagation strategy is to update the tree so that the selection strategy can make better choices. The changes that the backpropagation strategy makes depends on the information needed by the selection strategy. In most cases, this will be the expected value and visit count of the node. In this section, I will discuss the different backpropagation strategies and how I implemented them.

### 3.7.1 Averaging Backpropagation Strategy

This strategy does two things to every node it passes. First, it increments the visit count. Second, it updates the expected value to be equal to the average result of every simulated game that occurs further down the tree. It does this by calculating the weighted average of its children's expected values (where the weights are the visit counts). This strategy is fast and easy to implement and also quite effective.

### 3.7.2 Max Backpropagation Strategy

This strategy is similar to the previous strategy but with one main difference. When looking at a ChoiceNode, instead of calculating the weighted average, it uses the maximum expected value of its children.

The reasoning behind this is that, since it is the bot's choice, it would always choose the one with the highest expected value and so considering any other

option would be pointless. The averaging strategy takes this into account in a similar way. It relies on the selection strategy to select the node with the highest expected value most of the time. Then, when taking the weighted average, the highest expected value will contribute the most. However, the nodes with lower expected values will still make a small contribution and so the max strategy will always give slightly higher estimates than the averaging strategy.

In practice, I don't think there is a significant difference and both strategies seem to perform roughly the same. Because the max strategy overestimates the expected value and because I was already having some problems with overestimation, I decided not to use this strategy.

### 3.7.3 Averaging Var Backpropagation Strategy

The UCTVar selection strategy requires the variance of the expected values to make its decisions. This strategy is the same as the averaging strategy except that it also calculates the variance at each node.

Obviously, recalculating the variance from scratch every time (using the standard formula) would not be practical, a recursive formula is needed[3]. The formula I used is as follows [3].

$$\bar{x}_1 = x_1$$

$$S_1 = 0$$

$$\bar{x}_n = \frac{1}{n}((n-1) \cdot \bar{x}_{n-1} + x_n)$$

$$S_n = S_{n-1} + \frac{n}{n-1}(\bar{x}_n - x_n)^2$$

The standard deviation is calculated with:

$$s_n = \sqrt{\frac{1}{n-1} S_n}$$

Note that the variables $\bar{x}_n$ and $S_n$ are stored in the Node class and $s_n$ is calculated on demand.

[3] I know this from experience, in my very first attempt at creating this strategy, I did implement the naive version. Then when I implemented the recursive formula, the number of iterations per second approximately tripled.

I have observed very small rounding errors when using this formula, however they are usually only in the third or fourth significant figure so I believe they would have little effect.

## 3.8 Hand History Conversion

The next two sections regard the creation of the opponent models. One of the challenges in creating the opponent models was in collecting the training data to be used by Weka and converting it into a usable format. In this section I will describe how I did this.

### 3.8.1 Hand Histories

Poker Academy Pro provides the ability to record poker games and export them as text files. These game records are generally called hand histories. There is no exact standard and I have seen several different styles used in other programs. The following is an example of a hand history exported from Poker Academy Pro.

```
*****************************
Poker Academy Pro #133,388
Limit Texas Holdem ($1/$2)
Table SimpleBot Vs MCTSBot
May 07, 2011 - 13:57:15 (BST)

 1} David        (sitting out)
 5) MCTSBot *     $1,561  Ks Ts
 6) SimpleBot       $439  2s 9s

MCTSBot posts small blind $0.50
SimpleBot posts big blind $1
MCTSBot raises $1
SimpleBot folds

MCTSBot wins $2 uncontested
*****************************
```

The useful information is the actions that the players make, the cards that they have and the cards that are dealt onto the table. The rest of the information can be ignored.

### 3.8.2 ARFF

Ultimately, the plain text hand histories had to be converted into something that Weka can recognise. There were several possibilities including: using a database, using the Weka API or converting the data into Attribute Relation File Format (ARFF). I chose to put the data into ARFF because it seemed like the easiest option and because I wanted to use the GUI provided with Weka to help create the classifiers.

A .arff file is a plain text file which contains a list of instances sharing a set of attributes [13]. At the top is one or more @ATTRIBUTE tags followed by the name and type of the attribute. An attribute can be NUMERIC and take real or integer values or be NOMINAL and take one value out of a fixed list of values. Below that is the @DATA tag and everything below that is an instance.

Here is an example .arff file:

```
@RELATION time_spent_on_project

@ATTRIBUTE term {first, second, third}
@ATTRIBUTE hours NUMERIC

@DATA
first, 97.5
second, 61.5
third, 135.0
```

### 3.8.3 GameRecord and PlayerRecord

There is still the problem of converting the hand histories to ARFF. I decided to use an intermediate stage instead of going straight to ARFF because I knew that I would want to try many different ways of representing the data but that I only

needed to convert it once. Also, it takes a long time to convert hand histories from plain text format (around 1ms per game).

I created the GameRecord class to store the game records from the plain text hand histories. The GameRecord class stores the cards on the table, the stage that the game had reached and a list of PlayerRecords. The PlayerRecord class stores information about one of the players in the game. It stores their name, their hole cards, the rank of the best poker hand they can make (if applicable), their profit for the game and a record of all of the actions that they had taken. Both the GameRecord and PlayerRecord classes implement the Serializable interface and can be written to or read from a file.

The GameRecord class has a method called checkGame which checks various properties of the game to help find any errors or inconsistencies. It checks to see if things like the number of cards on the table is correct, or if there are any duplicate players, etc.

### 3.8.4 HHConverter

The conversion of plain text hand histories to GameRecords happens in a method in the HHConverter class. In this method, a BufferedReader is created around the input file and an ObjectOutputStream is created to save the GameRecords. Then the method starts reading the input file line by line.

When the method reaches the beginning of a game (indicated by a row of '*'s), it creates a new, empty GameRecord object. When it reaches a line defining an active player, it creates a new PlayerRecord object. It also takes note of the player's name, the player's cards and money and whether or not the player is the dealer. When it reaches a line stating that a player has performed an action, it looks up the correct player and adds the action to their list of performed actions. Finally, when it reaches a line stating that the game has reached a new stage, it saves the new table cards in the GameRecord.

The type of the line that is being examined is determined by using regular expressions and the matches method of the String class. The relevant information, such as the player name and bet size, is extracted using the substring and indexOf methods in the String class.

I have since learned of a much more elegant way to do this using the

java.util.regex.Matcher class. The Matcher class can check to see whether a String matches against a given regular expression and also find specified parts of the regular expression (called capturing groups) and return them. Here is an example of how this could work.

```
String re = "(\\S+) (?:bets|raises) \\$(\\d+(?:\\.\\d+)?)";
Pattern p = Pattern.compile(re);
Matcher m = p.matcher(inputLine);

if(m.matches()) {
    playerName = m.group(1);
    amount = Double.parseDouble(m.group(2));
    // Create a RaiseAction and add it to the player's action list.
}
```

Once the GameRecord object has been created, its checkGame method gets called. This is to check that the GameRecord represents a valid game and it should detect any errors that have been made. If the checkGame method fails then it will throw an exception. If it succeeds then the GameRecord will be written to a file with the ObjectOutputStream and the method will move onto the next game.

To test that this was all working as intended, I gave the method some sample hand histories and used the print method of the resulting GameRecords to view the information they contained. I then manually went through the original hand histories and looked for inconsistencies. A more thorough approach would have been to prepare a set of test cases and compare the results automatically. In retrospect, I think I should have done that.

### 3.8.5 WekaFormat

The hand histories are now in the form of GameRecords, however they are still not in a form that is usable by Weka. I created an interface called WekaFormat. This interface has a method called write which takes a GameRecord and writes it to a .arff file. It also has another method called writeHeader which writes the header information to the file (i.e. the @RELATION, @ATTRIBUTE and @DATA tags). The format in which the GameRecord is written is specified by the

class implementing the interface. I will discuss the specific classes implementing the WekaFormat interface in the next section.

The HHConverter class has a main method. When run, it calls the writeHeader method on its current WekaFormat. It then reads in the GameRecords from the previous step, one at a time, and writes them to a file using its current WekaFormat. This produces one .arff file, ready to be used to train a classifier.

## 3.9 Opponent Models

In this section, I will explain how I implemented the different opponent models used in the program and the different WekaFormats I tried.

Recall from the previous sections that there are two places where an opponent model can be useful.

1. In the simulation strategy to predict the result of a simulated game.
2. In the selection strategy to determine the proportion of time that each action is selected when considering an OpponentNode.

### 3.9.1 The Hand Rank Opponent Model

The original purpose of the hand rank opponent model was to predict the hand rank of an opponent, given their past actions in the current game and the cards on the table. In this subsection, I will discuss my original attempt and how I improved it.

The WekaFormat subclasses actually perform two functions. The first is to write a GameRecord to an .arff file, the second is to create a weka.core.Instance object from a GameState. An Instance object is used by every type of Weka classifier. Creating an Instance is very similar to writing to an .arff file because most of the information required is the same in both cases. Because of this, most of the code for generating instances is shared by the two different methods. There is a small problem of the different syntax used to create instances and write to a file. To get around this, I created an interface called OutputType, and two implementing classes, FileOutputType and InstanceOutputType. The FileOutputType takes

values and writes them to a file and the InstanceOutputType takes values and outputs a weka.core.Instance object at the end. This could be seen as another implementation of the strategy pattern.

My first attempt at representing the data in ARFF format was quite basic and slightly flawed. I counted the number of raise actions and call actions in each of the stages, included five strings representing the cards on the table and a number representing the hand rank of the player. One problem with this was that the chunks of data were much too coarse and Weka was unable to interpret anything from the betting order or the number of suited cards.

I was using the Weka Explorer to visualise the data and try out some of the different classifiers. I went through many different representations and classifiers before finally settling on one. My final format (called EverythingWekaFormat) worked slightly differently to my previous attempts. Instead of trying to predict the hand rank of the opponent's best hand, it attempts to predict the probability of winning, losing or drawing against the opponent. To do this, it uses a DistributionClassifier from Weka (specifically the NaiveBayes classifier). There were several reasons why I did this, the main one being that predicting the probability of winning is a much simpler task and it is less important if the result is slightly inaccurate.

The EverythingWekaFormat includes everything that could be considered useful, such as every action made at each stage of the game, the ranks of the table cards and information about the number of suited cards on the table after each stage. Also note that the hand rank of the other player in the game is included as one of the attributes. This is so that the strength of MCTSBot's hand is not ignored when simulating a game.

Using my final format and the NaiveBayes classifier, I was able to create a classifier which had a 65% success rate (using 10 fold cross validation on the training data). 65% might not seem sufficiently large but when the opponent model is used as part of the simulation strategy, the results of many simulations can provide a much more accurate picture.

It would be a little too time consuming to rebuild the classifier every time MCTSBot is launched. I created a main method in the Weka hand rank opponent model class which rebuilds the classifier from the training data and saves it to a file. Then, MCTSBot can just load that rather than having to rebuild it every time.

The hand rank opponent model[4] was the first opponent model that I implemented. I found that it provided a large boost to the performance of MCTSBot when played against SimpleBot, however it was not enough to allow MCTSBot to turn a positive average profit.

### 3.9.2 The Next Move Opponent Model

The next move opponent model calculates the probability distribution to be used by the selection strategy when selecting for an OpponentNode. Like the hand rank model, it also uses the EverythingWekaFormat and a DistributionClassifier. However, this time it is predicting the probability distribution for one of the opponents next actions rather than the result of the game.

A different classifier is needed for each action at each stage. There are 4 stages and up to 5 actions in each stage (although usually only about 1-3 actions are actually performed). This means than 20 different classifiers are needed. Each classifier only deals with the information which occurs before the specified action takes place and ignores everything afterwards.

I found that implementing the next move opponent model dramatically increased the performance of MCTSBot.

## 3.10 The GUI

A useful feature that I added to the program was a graphical interface for viewing up to date information about the progress of the algorithm. Before I implemented it, I would print debugging information to the standard output and error streams. However, this was quite inconvenient as it meant I had to quit Poker Academy and open the logs in order to view this information. My solution was to create a GUI that ran side by side with the Poker Academy program which would allow me to view the required information as it was being generated.

I created a class called GUI. It has a static method called initiate which creates a JFrame and adds several Components to it, including: a graph to display the expected values of the nodes over time; a JButton to force the algorithm

[4]Note that I kept the name even though the model no longer predicts the hand rank.

to restart; a JButton to force the algorithm to stop; a JTextField to change the thinking time; and a JLabel to display the current cards or other useful information.

The Graph class, which extends JPanel, has a method for updating the graph. It takes three ArrayLists containing the data points to be drawn. It then draws the lines on the graph, making sure to clip them to the drawing rectangle. It also calculates the vertical range which should be used so that all of the useful information is drawn to a sufficient level of detail.

For an example of what it looks like, see figure 3.1.

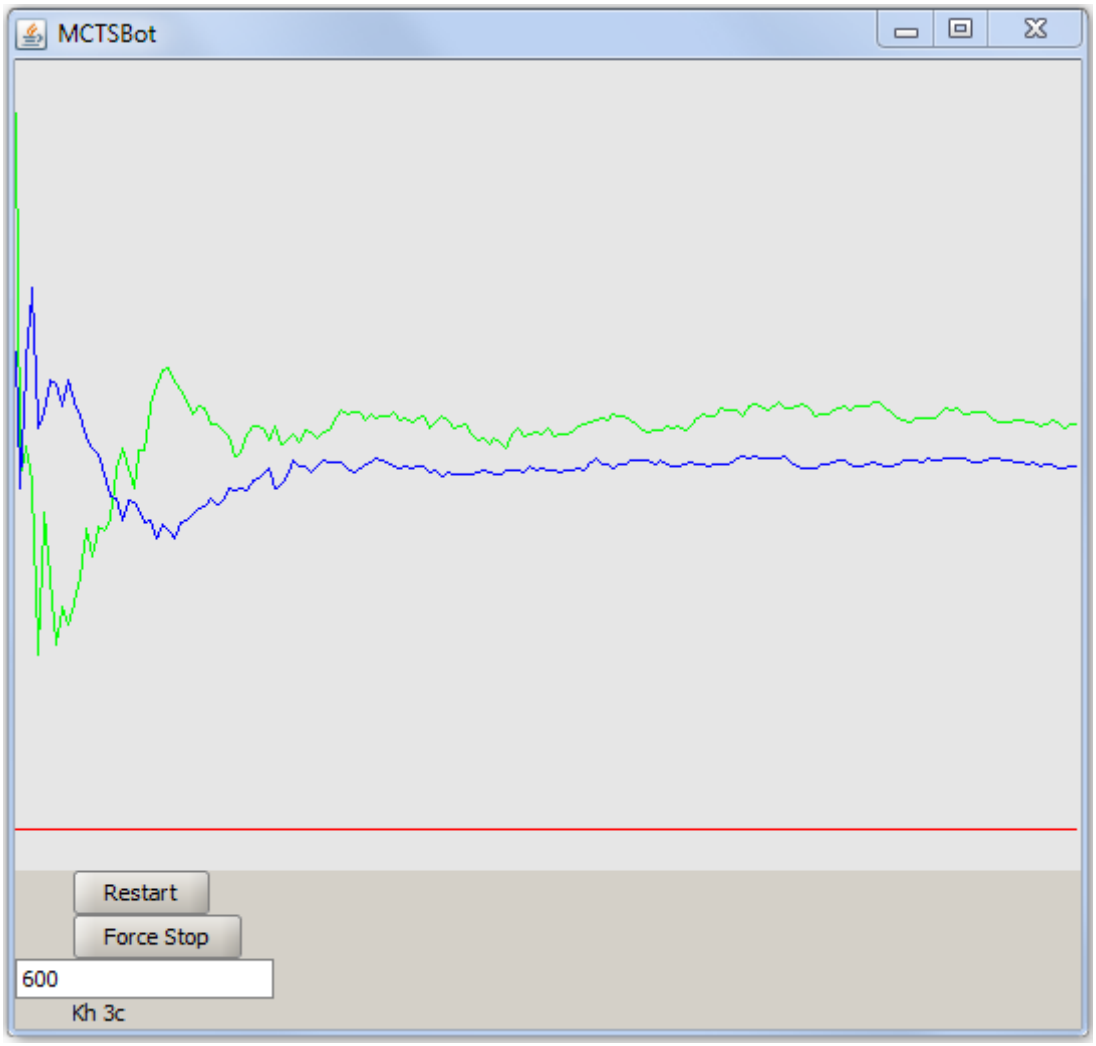


Figure 3.1: A GUI for viewing up to date information about the progress of the MCTS algorithm.

## 3.11 Summary

Overall, this was a very large project[5] and its performance matches my expectations. From preliminary testing, I have determined that MCTSBot can usually beat SimpleBot at a rate of about $0.20 per game. To put that in perspective, my earliest working version of the program (from Michaelmas term) was losing at about a rate of $0.25 per game.

In the next chapter, I will evaluate the program much more thoroughly in a variety of situations.

[5] Over 5000 lines of code in total and 58 classes.

# Chapter 4

# Evaluation

In this chapter, I will describe the experiments I performed to evaluate MCTSBot. There are 4 parameters that I will be altering, they are:

1. The thinking time available for the MCTS algorithm.
2. The presence/absence of the two different opponent models.
3. The selection strategy.
4. The number of opponents.

## 4.1 Experimental Setup

To perform most of these experiments, I played MCTSBot against SimpleBot in Poker Academy Pro and exported the hand histories. Because I had already made a class to convert the hand histories, I used it again to calculate the profit per hand and write it to a file. I created the graphs using a program called rlplot. Everything was done on the same computer for consistency[1].

Unless specified otherwise, the default parameters were:

[1]The specs of the computer are: Quad Core 2.4GHz CPU ; 3GB RAM; running Windows 7. However the program is only single threaded so only one core was being used. I also noticed that (one core of) the CPU was almost always at maximum utilization and so was probably the limiting factor.

- 600ms thinking time.
- Both opponent models (using the EverythingWekaFormat, Weka's Naive-Bayes classifier and training data from the 50,000 games of SimpleBot Vs. SimpleBot).
- The UCTVar selection strategy (with opponent model and constants $C_1 = 10$ and $C_2 = 0.1$) and the averaging var backpropagation strategy.
- The always call simulation strategy (with opponent model).
- A single opponent, SimpleBot.

I chose these values by trial and error, they are not necessarily optimal.

In these experiments, I will be measuring the average profit. The profit for one game is the difference between the amount of money MCTSBot has at the start and the amount of money it has at the end. The average profit is the mean of the profits for every game so far. It is measured in small bets per game[2]. I am measuring the average profit because I believe it provides a good indication of the overall performance of MCTSBot.

## 4.2 How Many Games?

There are many random elements when playing SimpleBot against MCTSBot. The main one being that random cards are dealt to each player and to the table. Also, in SimpleBot, random numbers are used frequently to decide which actions to take[3] and in MCTSBot, there are several random elements in both the simulation and selection strategies.

I needed to find out approximately how many games it is necessary to play in order to reach a stable result[4].

[2]A small bet is the amount a player can raise by in the first two stages of the game (it is equal to $1 in this case).

[3]In preflop, SimpleBot has a 5% chance to play any hand, even if it would have normally folded. In postflop, random numbers are generated and compared to calculated values representing the hand's strength, the pot odds and other values.

[4]If I had had more time then I would have investigated using statistical hypothesis testing [2].

To get a rough idea of how many games are necessary, I played out almost 65,000 games of SimpleBot Vs. MCTSBot using the default parameters. Here is a graph of MCTSBot's average profit over the course of those 65,000 games. Note that the dotted lines are drawn at two times the standard error from the mean.

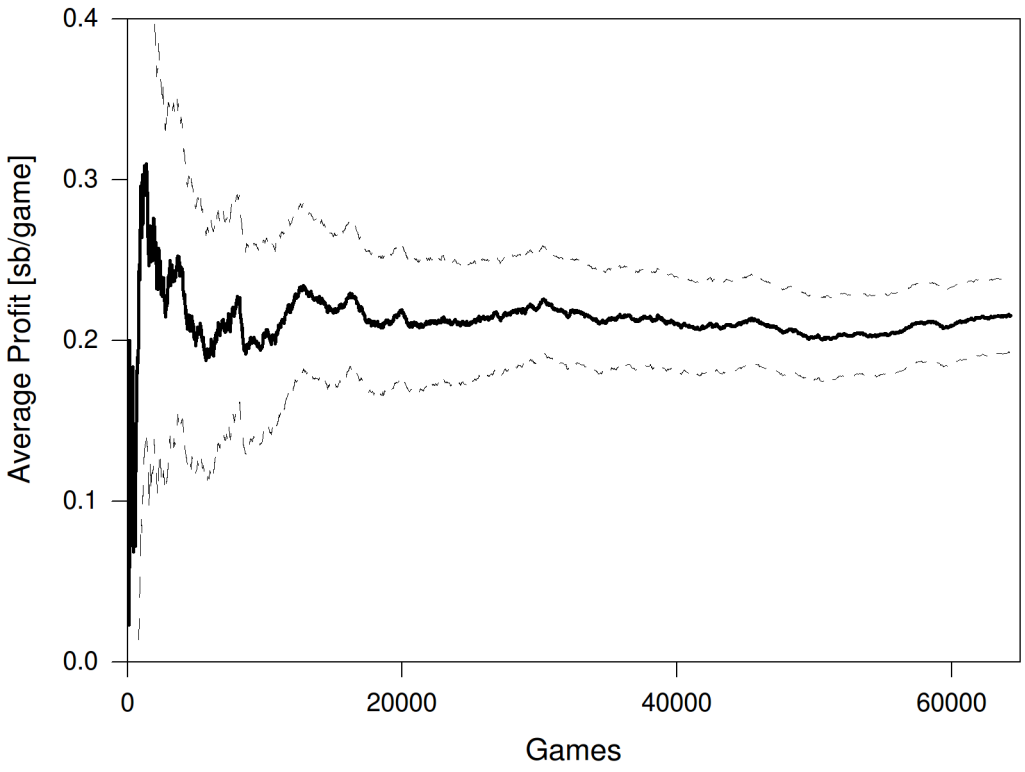


Figure 4.1: MCTSBot Vs. SimpleBot - 65,000 games with default parameters.

There seems to be little change after 20,000 games. Based on this, I planned to play out 20,000 games (for each variant) in the following experiments[5].

## 4.3 Varying Thinking Time

From the literature on MCTS and from my experience so far, I expected the average profit to increase as thinking time increases. I also expected the average

[5]Again, if I had more time then I would take a much more thorough approach. However, it can take over 24 hours to play out 20,000 games and so playing more is simply not practical.

profit to eventually level out. This is because no matter how good MCTSBot's decisions are, its profit is still limited by the rules of poker and by SimpleBot's play style.

To test the effects of thinking time on performance, I ran five trials of 20,000 games each. The only variable I changed was the amount of thinking time given to the MCTS algorithm. I used the default thinking time (600ms) as well as four times the default (2400ms), one quarter of the default (150ms), one sixteenth of the default (38ms) and one sixty fourth of the default (9ms). I chose those values because I think they represent a wide range of abilities for the MCTS algorithm.

Here is a graph showing the average profit for the five different thinking times. The error bars are drawn at two times the standard error from the mean. Also note that time is on a log scale.

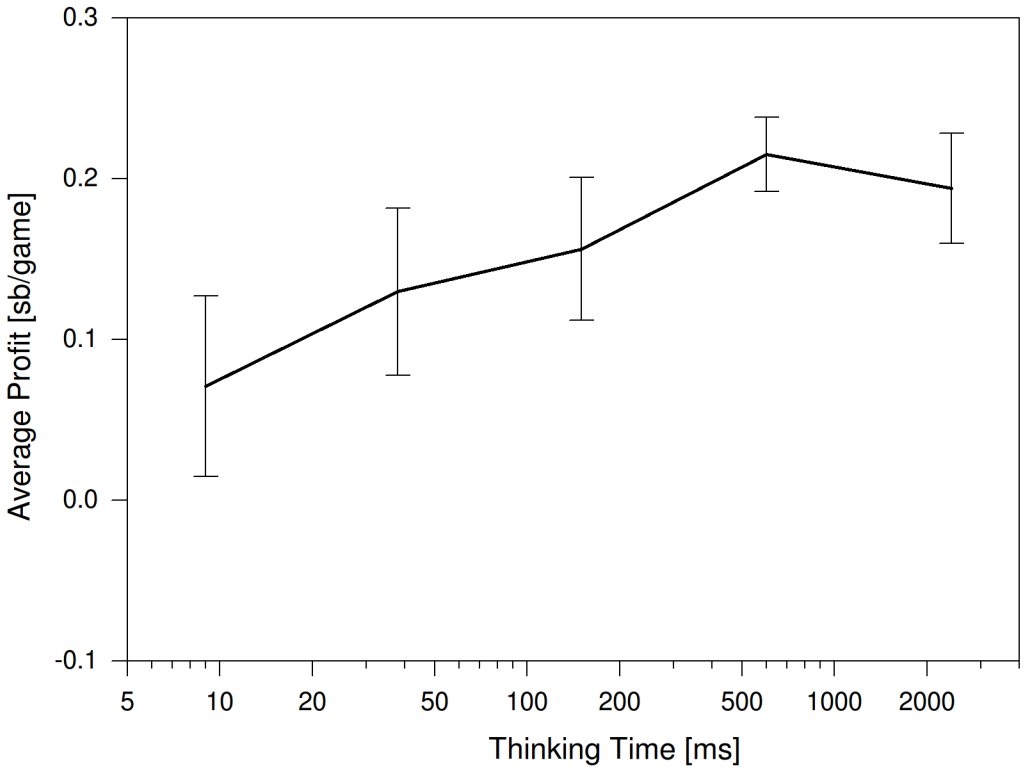


Figure 4.2: Average profit of MCTSBot against SimpleBot with different thinking times.

I think it is clear from these results that increasing the thinking time has a

positive effect on the average profit.

At 2400ms thinking time, the average profit is slightly lower than at 600ms. Given that both means lie within the error bars of the other, it is most likely due to chance[6].

## 4.4 Varying Opponent Models

One of the original questions I wanted to answer was: how much of an effect do the opponent models have? To find out, I played MCTSBot against SimpleBot using all four combinations of the two opponent models, that is: both of them, just the hand rank opponent model, just the next move opponent model and neither of them.

Note that when I say an opponent model is not being used, I mean that I am using the basic model instead of the one using Weka classifiers. The basic hand rank opponent model is the one which randomly deals the opponent some hole cards and sees who has the best hand. The basic next move opponent model is the one which assigns a probability of $\frac{1}{3}$ to each of the possible moves[7].

Here is a bar chart displaying the results, the error bars are drawn at two times the standard error from the mean[8].

[6]However, it could also be caused by something else. If I had more time, I would investigate this further.

[7]A basic model which returned different probabilities might perform slightly better. I did briefly experiment with using different probabilities, however it still did not perform very well.

[8]Note that the no-models and hrom-only results were attained after only 5000 games. This was due to time constraints.

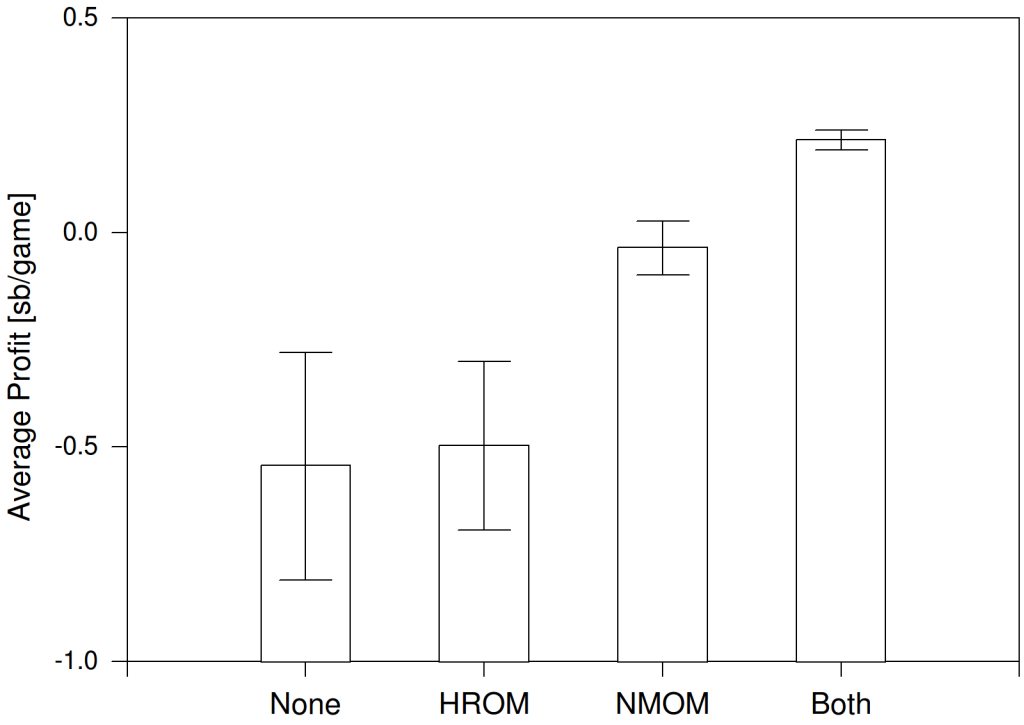


Figure 4.3: Average profit of MCTSBot against SimpleBot with different combinations of opponent models.

The graph shows that using both opponent models is far superior to any other combination. In fact it is the only combination which actually makes a positive average profit. Using only the next move opponent model is the next best, almost managing to break even. The other two combinations both perform equally poorly.

I think these results show that both opponent models clearly benefit the program and that the next move opponent model is more beneficial than the hand rank opponent model.

## 4.5 Varying Selection Strategies

When dealing with ChoiceNodes, there are two possible selection strategies, UCT and UCTVar. Here is a bar chart showing the average profit for each strategy. Note that I used $C = 10$ for the UCT strategy and $C_1 = 10$ and $C_2 = 0.1$ for

the UCTVar strategy. The error bars are drawn at two times the standard error from the mean.

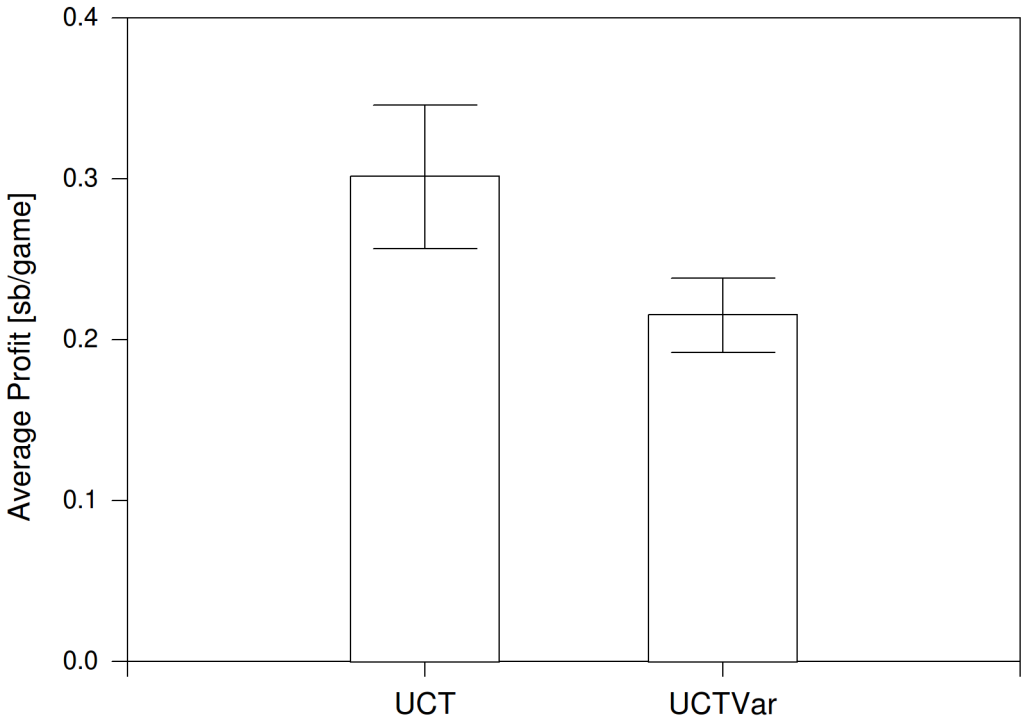


Figure 4.4: Average profit of MCTSBot against SimpleBot with the UCT and UCTVar selection strategies.

The results clearly show that the UCT strategy is performing significantly better than the UCTVar strategy. This is exactly the opposite of what I expected[9].

The reason for this might be that I am not using the best possible constants. I may also have implemented the UCTVar strategy incorrectly or there could be bugs in the code. All I can say is that, in my implementation, the UCT strategy is superior to the UCTVar strategy.

[9] I am not sure how to explain this, especially as I remember the UCTVar strategy performing better in early testing. I think I might have inadvertently fixed a crippling bug in between testing the two strategies, which made me think that the UCTVar strategy was increasing the performance.

## 4.6 Varying the Number of Opponents

Another one of the original questions that I wanted to answer was: what effect does increasing the number of opponents have? There are several things to consider. The first is that, when there are more opponents, there is a lower probability that MCTSBot will have the best hand[10]. The second is that the MCTS algorithm does not explicitly take into account the fact that when an opponent does stay in the game, they are more likely to have a better hand than if they had made the same actions in a game with fewer players. This is because the opponent model was trained on data from two player games only. Also, since there are more players in the game, there will be a greater number of branches in the game tree and so the MCTS algorithm will have less time to search through each one.

On the other hand, you could argue that MCTSBot has more opportunities to win money through superior play. However, I was still expecting MCTSBot's average profit to decrease as the number of opponents increases.

The following graph shows the average profit of MCTSBot when played against one, two, three and four separate instances of SimpleBot in Poker Academy Pro. Error bars are drawn at two times the standard error from the mean.

[10] In fact you would expect the probability to be $\frac{1}{n}$ where $n$ is the total number of players in the game.

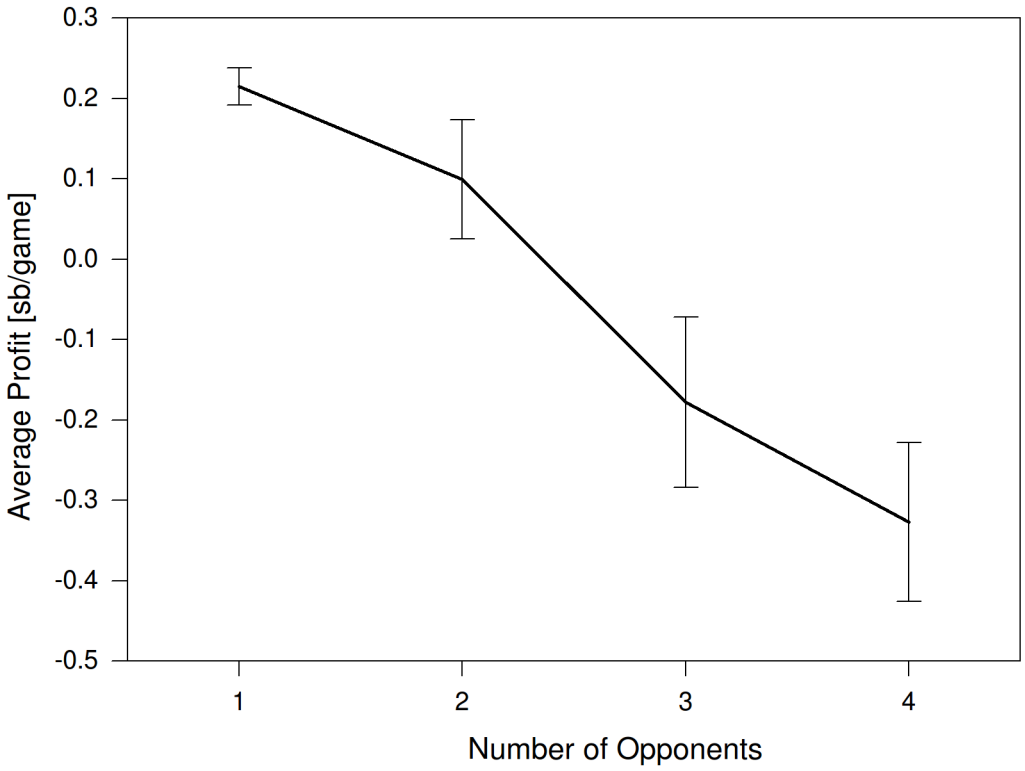


Figure 4.5: Average profit of MCTSBot
when played against multiple instances of SimpleBot.

As you can see, the average profit steeply decreases as the number of opponents increases. MCTSBot is able to make a positive average profit when played against one or two opponents but not when played against three of four.

One of my original success criteria was for MCTSBot to be able to make the most money when played against three separate instances of SimpleBot. In the above experiment, MCTSBot did not achieve this. However, I now know that the default settings I have been using are not optimal. I wanted to complete all of my original success criteria so I attempted to tweak the parameters of MCTSBot to increase its performance.

I decided to use the UCT selection strategy (with $C = 10$) instead of the UCTVar selection strategy. I also gave the MCTS algorithm 1000ms thinking time instead of 600ms. The following graph shows the average profit of MCTSBot and all three of the instances of SimpleBot.

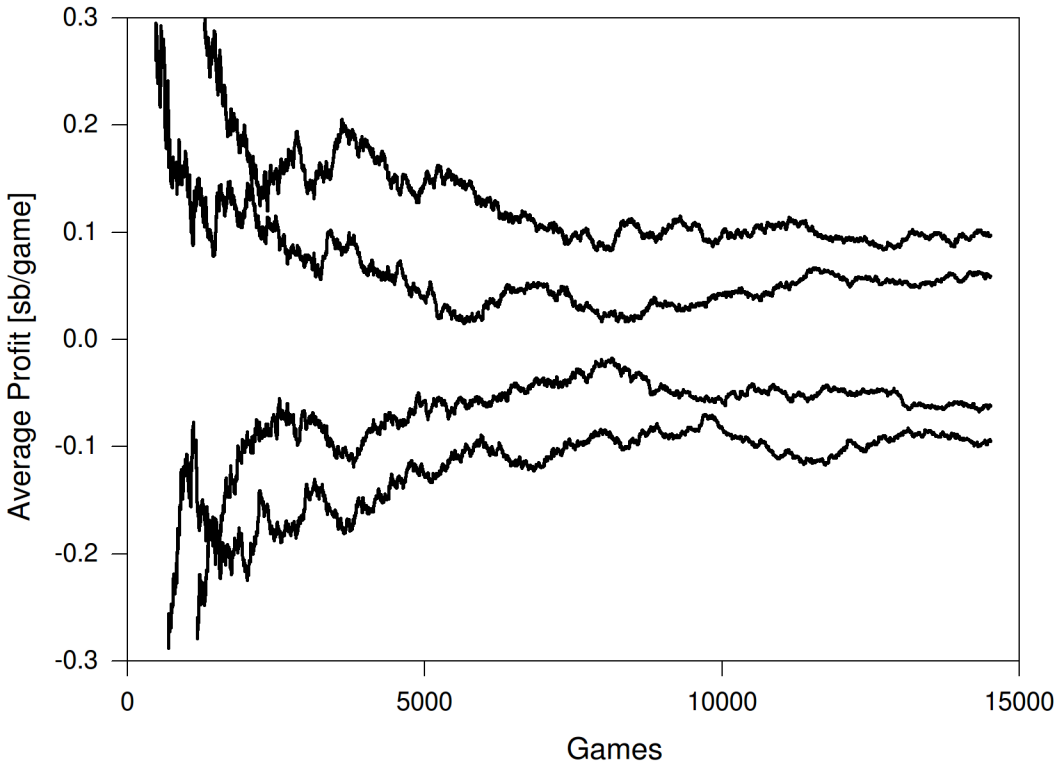


Figure 4.6: Average profit of MCTSBot and three instances of SimpleBot over the course of about 15,000 games.

The top line is MCTSBot's average profit, the three lines below that are the average profits of the three instances of SimpleBot[11].

An interesting thing to note is that the SimpleBot which was sitting to the right of MCTSBot, seems to be doing much better than the other two instances of SimpleBot. I have noticed this every time I have played games with multiple opponents. I think this could be due to a situation in which MCTSBot has a bad hand but gets drawn into the next round with the player to its right just because it is on the big blind.

[11]In my opinion, this shows that MCTSBot can make the most money when played against three instances of SimpleBot. If I had more time, I would play more games and try to show this more conclusively.

## 4.7 Other Opponents

I originally chose SimpleBot because it was simple yet still challenging. However, I think it would be interesting to see how my program fares against some other poker bots.

Here is a table showing the average profit of MCTSBot when played against the opponents included in Poker Academy Pro[12].

| Opponent | Average Profit |
|---|---|
| Simbot | +0.39 sb/game |
| Jagbot | +0.22 sb/game |
| Pokibot | -0.75 sb/game |
| Sparbot | -0.75 sb/game |
| Vexbot | -1.75 sb/game |

MCTSBot appears to perform well against Simbot and Jagbot, poorly against Pokibot and Sparbot and extremely poorly against Vexbot. I think this shows that, when designing a poker bot, it is important to test against a wide variety of opponents because some strategies may be effective against some opponents and ineffective against others.

Note that MCTSBot is using the default parameters, including the opponent models trained on the SimpleBot Vs. SimpleBot hand histories. This means that a lot of MCTSBot's strategy is specific to SimpleBot. I have observed that Vexbot in particular seems to exhibit behaviours which SimpleBot does not. For example, sustained attempts at bluffing. I think that if I were to implement a more general opponent model, it would increase MCTSBot's playing ability when facing other opponents.

[12]Note that due to time constraints, I was only able to play about 3000 games for each opponent.

# Chapter 5

# Conclusion

## 5.1 Results

My program, MCTSBot, has proven to be a competent poker player when facing simple opponents. It can consistently beat SimpleBot in a two player match. It can also beat three separate instances of SimpleBot in a four player match.

I have also demonstrated the effects on performance of varying factors such as the thinking time, the presence of the opponent models, the selection strategy and the number of opponents.

## 5.2 Future Work

I think the biggest weaknesses of MCTSBot are its limited opponent models. Currently, they are too specific to SimpleBot and too specific to games with only two players. This means that MCTSBot will often make poor choices when facing other opponents or playing in games with more than two players.

I would like to expand the opponent models for use against other opponents. I also think it would be interesting to create an opponent model with the ability to adapt to unknown opponents or maybe one which could learn from its own mistakes.

## 5.3 Summary

In this project, I have successfully built a working poker bot. I implemented the Monte Carlo Tree Search algorithm with several alternative strategies and I created two simple opponent models.

I stuck to my original plan quite closely although it took a lot longer than I had anticipated. If I could go back and change one thing, I would have chosen opentestbed instead of Poker Academy. Poker Academy is not ideal for evaluating poker bots and it does not provide a way to control the random elements in poker.

However, I am delighted with the results I have achieved and with the success of the project as a whole.

# Bibliography


[1] http://www.outflopped.com/questions/286/obfuscated-datamined-hand-histories.

[2] http://en.wikipedia.org/wiki/Statistical_hypothesis_testing.

[3] H. R. Biesel. Recursive Calculation of the Standard Deviation with Increased Accuracy. *Chromatographia*, Vol. 10(No. 4):173–175, April 1977.

[4] Guillaume Chaslot. Monte-Carlo Tree Search, 2010.

[5] Aaron Davidson. Opponent Modeling in Poker: Learning and Acting in a Hostile and Uncertain Environment. Master's thesis, University of Alberta, 2002.

[6] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-Carlo Tree Search in Poker using Expected Reward Distributions, 2009.

[7] D. Billings et al. Approximating Game-Theoretic Optimal Strategies for Full-scale Poker, 2003.

[8] G. E. H. Gerritsen. Combining Monte-Carlo Tree Search and Opponent Modelling in Poker. Master's thesis, Maastricht University, 2010.

[9] BioTools Inc. Meerkat API 2.5. http://www.poker-academy.com/community.php, 2006.

[10] BioTools Inc. Poker Academy Pro 2.5. http://www.poker-academy.com/, 2006.

[11] Michael Bradley Johanso. Robust Strategies and Counter-Strategies: Building a Champion Level Computer Poker Player. Master's thesis, University of Alberta, 2007.

[12] Francois Van Lishout, Guillaume Chaslot, and Jos Uiterwijk. Monte-Carlo Tree Search in Backgammon, 2007.

[13] University of Waikato. ARFF. `http://weka.wikispaces.com/ARFF`.

[14] University of Waikato. Weka. `http://www.cs.waikato.ac.nz/ml/weka/`.

[15] Marc Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating Opponent Models with Monte-Carlo Tree Search in Poker, 2010.

[16] schatzberg.dan@gmail.com and bluegaspode@googlemail.com. opentestbed. `http://code.google.com/p/opentestbed/`.

[17] Terence Conrad Schauenberg. Opponent Modelling and Search in Poker. Master's thesis, University of Alberta, 2006.

[18] A. A. J. van der Kleij. Monte Carlo Tree Search and Opponent Modeling through Player Clustering in no-limit Texas Holdem Poker. Master's thesis, University of Groningen, 2010.

# Appendix A

# Project Proposal

David Moody
Christ's College
dm484

Diploma in Computer Science Project Proposal

## Monte Carlo Tree Search in Texas Hold 'em Poker

17 October 2010

**Project Originator:** Sean Holden

**Resources Required:** See attached Project Resource Form

**Project Supervisor:** Sean Holden

**Director of Studies:** Marcelo Fiore

**Overseers:** Andy Rice and Jean Bacon

## Introduction and Description of the Work

There are two types of poker playing program. The first type attempts to approximate a perfect strategy using game theory. While this might be the best way to play if your opponents are also playing with an optimal strategy, it can't exploit the possible weaknesses of your opponents. The second type of program can exploit the weaknesses in its opponents' strategies. It does this in order to try and maximize its total income and thus outperform the first type. In my project I will be looking at the second type of program.

In the specific variety of poker which I will be using, Texas Hold 'em, each player is first dealt two cards which only they can see. There is then a round of betting in which each player in turn is given the choice to either: raise, where they place an amount of money into a shared "pot"; call, in which case they must put into the pot the same amount of money as the last person in the current round who raised (possibly nothing); or fold, in which case they forfeit the game and lose any chance to win the money in the pot. After this round of betting, three shared cards are dealt which everyone can see. Then there is another round of betting, another shared card, another round of betting, another shared card and a final round of betting. At this point, it more than one player is still in the game then the player who can make the best "poker hand" out of their cards and the shared cards wins the money in the pot. If all but one player folded before this point then the remaining player wins the money in the pot.

There are two main varieties of Texas Hold 'em. In the first (no-limit), you can raise any amount that you want. In the second (limit), you can only raise by a fixed amount. While no-limit Texas Hold 'em might also make a good game for this project, I will be using the limit variety.

Poker is usually played with somewhere between 2 to 10 players. When you have only 2 or 3 different players it is feasible to do a complete search of the game tree in order to find an estimate of what you think the optimal strategy would be. However, when you are playing with more than 3 players the game tree becomes much much larger. In a game with 10 players it would be impossible to do a complete search of the tree.

Thus we need to use an algorithm that can approximate the best choice without needing to search the whole tree. Monte Carlo Tree Search can do this by randomly simulating many games and using the results of those simulations to

construct a partial game tree with estimates for the expected values of the nodes.

The MCTS algorithm consists of 4 main stages:

1. Selection: In this stage, starting at the root, you need to select which branch of the tree you want to explore further. I will be investigating multiple selection strategies including random selection, Upper Confidence Bound selection and selection using a complex opponent model to predict your opponent's most likely action.

2. Expansion: Once you reach a leaf in the stored partial tree you need to expand the tree by adding new nodes. You simply need to decide if the node of the partial tree is also a node of the full game tree and if not you need to add children representing the possible actions that could occur next.

3. Simulation: Starting from one of the newly added children, you need to simulate the completion of the game. There are several ways in which you could do this including random simulation or simulation based on more complex heuristics. I will investigate multiple simulation strategies. Once you have simulated the game to completion you need to calculate the expected value of the game state. The opponent model will be used to attempt to estimate what the opponents' cards may be based on their previous actions, thus allowing you to estimate the expected value of the game state.

4. Backpropagation: Once you know the expected value of the simulated game, you need to update the expected values of the nodes in the path that you took to include this information.

These 4 steps are repeated to build up the game tree until there is no more time left for computation. An action is then chosen, usually based on the highest expected value of the children of the root node.

In order to take advantage in the weaknesses in our opponents' strategies we also need an opponent model capable of predicting the following two things:

1. The probability that an opponent will perform a specific action given their actions up to this point in the game and the shared cards currently on the table.

2. The probability that an opponent has specific cards in their hand given their actions throughout the current game and the 5 shared cards on the table.

I will use Weka's [1] algorithms to create regression models for each of these.

Training data for the first model will be take the following form:
$A_1, C_1, C_2, C_3, A_2, C_4, A_3, C_5, A_4$
Where $A_i$ is the action (fold, call or raise) taken by the opponent at step i and $C_j$ is the $j_{th}$ shared card to be dealt. I may experiment with different ways of representing the cards.

I think I would need to have separate regression models for predicting each $A_i$. i.e. the training data for each $A_i$ would look like this:
$A_1, C_1, C_2, C_3, A_2, C_4, A_3, C_5, A_4$
$A_1, C_1, C_2, C_3, A_2, C_4, A_3$
$A_1, C_1, C_2, C_3, A_2$
$A_1$

This would make predicting the first action rather trivial, however it might also be a rather poor way to predict the first action. I might investigate how including other fields, such as the number of players to the right and to the left of the opponent or the position of the player who first raised, affects the ability to predict the first action of an opponent.

For predicting what hand the opponent might have we could use training data of this form:
$A_1, C_1, C_2, C_3, A_2, C_4, A_3, C_5, A_4, H_1, H_2$
Where $H_1$ and $H_2$ are the cards held by the opponent.

This also raises the question of where I will get the records of played games (aka hand histories) from which I can extract this training data. There is 70GB of obfuscated (names of players have been changed) hand histories available at source [2] which I think will be sufficient and I may be able to find more if required.

Unfortunately this data is not in a form that will be readable by Weka so I will need to write tools able to convert it into the Attribute-Relation File Format which Weka uses.

It will be possible to create training data for the first type of model from every action which takes place in the games. However for the model which predicts which cards the opponent holds, only games in which two or more players remained after the final round of betting can be used.

Once I have a program capable of playing poker I will need to test it to see how well it can play against other poker bots. I will integrate my program with the Poker Academy Pro software and test it against the poker bots which come with the software. I will consider the project a success if it is able to consistently beat a program called SimpleBot [4] which only considers its own hand when determining how to play. I will also be testing my program against the other poker bots available with the software, however being able to beat them will not be one of my success criteria because I don't want to be directly competing with the creators of the other poker bots. (Many of the existing poker bots were created by teams of researchers with much more experience than me.)

In addition to testing my program against other programs, I can test different versions of my program against each other to investigate how effective different selection strategies/opponent models are.

For extensions of the project, if I have time remaining after implementing what I have so far described, I would like to look at improving the selection and simulation strategies of the MCTS algorithm and improving the prediction capabilities of the regression models. If I still have time after that I will look at the possibility of changing the opponent model to be able to record statistics about specific players it has played against and alter its decisions based on those observations.

# Resources Required

- Algorithms from the Weka toolbox (a free collection of algorithms under the GNU General Public License) [1].
- Records of online poker games (aka hand histories) available online for free [2].
- Poker Academy Pro and the Meerkat API for interaction with other poker bots [3]. Poker Academy Pro is not free but I don't mind paying the $85 required to purchase it. The software is only needed when testing the

program against other poker bots so it won't be needed to understand or test the majority of the project.

- The core Java packages and the eclipse IDE.
- The use of my own computer.

## Starting Point

I already play Texas Hold 'em Poker regularly and I think I'm familiar enough with the game to be attempting this project.

The Artificial Intelligence I course from part IB contains information about best-first search techniques and machine learning techniques which will be very useful throughout the project.

The Java courses and Algorithms courses from part IA and IB will also be helpful.

## Substance and Structure of the Project

The aim of this project is to create a poker playing program capable of playing Limit Texas Hold 'em in games with more than 3 players. The program will use Monte Carlo Tree Search to partially explore the game tree and machine learning techniques to create an opponent model capable of predicting the probabilities of an opponents next action. The program will be integrated with the Meerkat API which will allow it to play against other poker bots included with the Poker Academy Pro software.

The project consists of 9 main stages:

1. Further research into opponent modeling, MCTS and different selection strategies for it, Poker Academy Pro and existing poker playing programs for it, the Weka toolkit, availability/quality of hand histories and how to convert them to ARFF, etc.

2. Creation of the basic framework of the program and integration with the Meerkat API. Also deciding upon how to abstract away some of the unnecessary details in poker.

3. Implementation and testing of the basic MCTS algorithm as described in the Introduction and Description of the Work section.

4. Collection and consolidation of hand histories into Attribute-Relation File Format.

5. Using Weka to create regression models for calculating the required probabilities and integrating those models into the opponent model.

6. Integration of the opponent model with the MCTS algorithm.

7. Evaluation and testing of the program against SimpleBot and other programs using Poker Academy Pro. Investigation into how effective MCTS with an opponent model is when compared to just MCTS. Investigation into the effects of varying the available search time and increasing the number of opponents.

8. Possible extensions such as changing the opponent model to adapt to specific players and trying out and comparing different selection/simulation strategies in the MCTS algorithms. (I don't want to say exactly what I'm going to be doing right now because I don't know how much time I will have and I don't yet know what would be the most interesting to do.)

9. Writing the dissertation.

# References

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

[2] `http://www.outflopped.com/questions/286/obfuscated-datamined-hand-histories`
This is actually for no-limit Texas Hold 'em but I am reasonably confident that I will be able to convert it for use in this project. If not, I may be able to find other sources as well. Also, even if the data does turn out to be totally unusable and I cant find anything else it wont have too much of an affect the rest of the project so Im not concerned.

[3] http://www.poker-academy.com/community.php
Information on the Meerkat API is at the bottom of the page.

[4] http://code.google.com/p/opentestbed/source/browse/src/bots/demobots/SimpleBot.java

[5] https://lirias.kuleuven.be/bitstream/123456789/245774/1/acmlpaper.pdf

[6] http://www.personeel.unimaas.nl/m-ponsen/pubs/Ponsen-MCTSMODEL-IDTGT10.pdf

[7] http://www.unimaas.nl/games/files/msc/Gerritsen_thesis.pdf

## Success Criteria

The following should be achieved:

1. Successful implementation of the MCTS algorithm.
2. The creation of regression models using Weka.
3. The program should be able to play Limit Texas Hold 'em against other programs using the Poker Academy Pro software.
4. The program should be able to make more money when played against SimpleBot in a run of 1000 games of 2-player Limit Texas Hold 'em.
5. The program should be able to make more money than any of its opponents when played against 3 instances of SimpleBot in a run of 1000 games of 4-player Limit Texas Hold 'em.
6. It should be demonstrated how much of an increase in performance a complex opponent model brings to the MCTS algorithm as opposed to a random opponent model.

7. It should be demonstrated how increasing the number of opponents and how varying the amount of search time affects performance.

I will also test my program against some of the other poker bots available with Poker Academy Pro and I might even try my program against other human players online. However, the success of my program in these situations should not be considered as success criteria for the project.

## Timetable and Milestones

I've broken the project up into blocks of work, each of which should take me around two weeks to complete. I think I might be a bit optimistic about how quickly I'll be able to complete the work so I've left myself plenty of time near the end. On the other hand, if I am able to finish the work sooner than I have estimated then there is still plenty that I could add to the project.

*Start working on project - 23 Oct*

1. Further research into: opponent modeling, MCTS and different selection/simulation strategies for it, Poker Academy Pro and existing poker playing programs for it, the Weka toolkit, availability/quality of hand histories and how to convert them to ARFF, etc.
2. Start work on designing the Java program. Draw diagrams indicating how the different components will interact. Start work on creating the basic framework of the program using the classes from the Meerkat API. Decide how to abstract away some of the unnecessary details in poker. (This block could be run concurrently with block 1.)
3. Start work on the MCTS algorithm. Create classes for the different kinds of nodes in the game tree. Implement the selection, expansion, simulation and backpropagation steps of the MCTS algorithm using simple selection and simulation strategies. Create a very basic opponent model (which assumes that the opponents always play randomly and has random cards) and use it to implement the simulation and selection steps. Test the MCTS algorithm.

- Milestone: Have a program that would be capable of playing poker (though not necessarily well).

*End of first term - 3 Dec*

5. Test the program against SimpleBot in Poker Academy Pro. See how well it does and see whether or not/how many iterations it takes to beat SimpleBot as it is.

- (Possible) Milestone: Have my program beat SimpleBot.

6. Start collecting records of poker games played by humans. Write tools to convert these records into ARFF. Play around with Weka.
7. Start work on the opponent model. Use training data and Weka to create regression models capable of predicting the required probabilities.

- Milestone: Create at least one regression model using Weka which can produce reasonable sounding predictions.

*Start of second term - 18 Jan*

8. Investigate different methods for creating the regression models. Integrate the regression models into the opponent model. Test my program against SimpleBot.

- Milestone: Have my program beat SimpleBot.

9. Review work done so far. Write progress report/practice presentation.

- Milestone: Hand in progress report/give presentation.

*Progress report deadline - 4 Feb / Progress report presentation - 15 Feb*

10. I don't know whether I'll still be on schedule by this point, I'm guessing probably not. I'll either use this time to catch up or start work on the extensions mentioned previously.

11. Continue working on extensions for the project. Maybe start thinking about writing the dissertation. (I aim to have finished pretty much everything except for the dissertation by the end of the second term.)

*End of second term - 18 Mar*

12. Start writing dissertation.
13. Continue writing dissertation/revisit areas of the project which I might have neglected.
14. Continue writing dissertation.

- Milestone: Finish a complete draft of the dissertation.

*Start of third term - 26 Apr*

15. Finish final touches on dissertation. (I will be spending the rest of the term focusing on the exams.)

- Milestone: Hand it in!

*Dissertation deadline - 20 May*