Analysis of Monte Carlo Techniques in Othello

Ryan Archer

This report is submitted as partial fulfilment
of the requirements for the Honours Programme of the
School of Computer Science and Software Engineering,
The University of Western Australia,
2007

Abstract

Gaming AI is a field that has become very popular recently, with milestones such as beating the chess grandmaster gaining considerable public interest. Some AI techniques have been very successful for some games but not so much for other types. A Monte Carlo approach uses many random game simulations to help choose moves that have a higher chance of winning. The algorithm that we're using is called UCT and combines Monte Carlo simulations with a tree search to balance the 'exploitation vs. exploration' problem. This technique is relatively new but is gaining popularity, partly due to a successful implementation used in the game Go. In this project a Monte Carlo agent using UCT is tested using the game Othello. We present experiments that compare the effects of increasing the number of simulations, using different heuristic functions and using an opponent model. The number of simulations has been found to be critical, but by using very simple heuristic functions performance can be greatly improved. Our experiments also highlight the importance of the opponent model.

Keywords: Monte Carlo, Othello, Artificial Intelligence
CR Categories: I.2[Computing Methodologies]: Artificial Intelligence, I.2.1 [Games], I.6.8 [Monte Carlo]

Acknowledgements

I would like to thank my family, as without their constant support throughout my entire education I would never have made it here. Dave deserves extra credit for proof reading my dissertation under stressful, last minute conditions.

The friendships that have lasted though uni and even some since high school have earned a special mention for maintaining my (hopefully) level head under stressful times. I especially appreciated Wednesdays in the Tav and Thursday nights, as great honours free zones. A special thanks to Conor for all her encouragement, support, motivation and great company.

My supervisor, Dr Luigi Barone, deserves special thanks for his ongoing support and encouragement throughout this year. He has made the weekly meetings thoroughly enjoyable and a good laugh was always guaranteed. Phil Hingston's help with possible ideas was much appreciated and finally thanks to Simon Lucas for the use of his Othello code.

Contents

Abstract **ii**

Acknowledgements **iii**

1 Introduction **1**

2 Othello **3**

2.1 History and Rules 3

2.2 Basic Othello Strategy 5

2.2.1 Stable Disks 5

2.2.2 Mobility 5

2.2.3 Parity 6

2.3 Previous Othello Players 6

2.3.1 Iago (1982) 6

2.3.2 Bill (1990) 6

2.3.3 Logistello (1994) 7

2.4 Roxanne 7

2.5 Motivation for using Othello 8

3 Monte Carlo **9**

3.1 Game Trees and Minimax 9

3.2 Basic MC algorithm 10

3.3 UCB1 12

4 Proposed Approach **14**

4.1 Project Aims 14

4.2 Implementing UCT 14

4.3 Heuristic Functions 18

4.3.1 Roxanne . . . 18
4.3.2 Mobility . . . 18
4.3.3 Pot.Mob. . . . 19
4.3.4 CPM . . . 19
4.3.5 CPMR . . . 19
4.4 Opponent Modelling . . . 20
4.4.1 Combining Heuristic Functions . . . 20
4.4.2 Implementing Noise in the Opponent Model . . . 21

5 Experiments and Results 22
5.1 Environment . . . 22
5.2 AgentX vs. AgentY . . . 22
5.3 MC(AgentX) vs. AgentY . . . 23
5.4 MC(AgentX, CPMR) vs. AgentX . . . 24
5.5 MC(AgentX) vs. MC(AgentY) . . . 25
5.6 Equal Time Allowed for Each Move . . . 27
5.7 Comparison of Two MC Agents at Varying Times per Move . . . 29
5.8 The Effect of Noise . . . 30
5.9 Monte Carlo Strength Against Humans and Minimax . . . 32

6 Conclusion 34

A Original Honours Proposal 36

List of Tables

2.1 The priorities Roxanne gives to each position on the board. 8

5.1 Agents vs. Agents 23
5.2 MC(h) at 50 Simulations vs. each Agent 23
5.3 MC(h) at 50 Simulations vs. each Agent with the addition of an MC(Agent, CPMR) vs. Agent row. 25
5.4 MC($h1$) vs. MC($h2$) at 50 Simulations. 26
5.5 MC($h1$) vs. MC($h2$) with equal time per move. 27
5.6 MC($h1$) vs. MC($h2$) with equal time per move, but the time doubled. 28
5.7 MC(h) with equal time per move vs. each Agent 29
5.8 MC(Random) v MC(CPM) at different number of simulations . . 30
5.9 MC(Roxanne) v MC(CPM) at different number of simulations . . 30
5.10 MC(AgentX) with noise vs. AgentX 31
5.11 Monte Carlo Agents (CPM) with noise vs Agents 33

A.1 Game Complexity 38

List of Figures

2.1 My Othello Board. 4

4.1 UCT. Root node and children. 15
4.2 UCT. Root node and children after one random simulation. 16
4.3 UCT. Root node and children after three random simulations. . . 16

5.1 Non-Linear relationship between playing strength and the number of simulations 31
5.2 Noise 32

CHAPTER 1

Introduction

The computer games industry is big business with a study in 2004 showing that 60% of all Americans aged six or older play video games [13]. Not only are we interested in games for consumerist reasons but also for the benefits games and gaming techniques have on society. For example, experts point out that games are an important stepping stone for kids into the world of technology [11]. Students who play computer games tend to be more comfortable with the technology and more adept at using it. Computer games fill the role as being an incubator for many technologies that drive the usefulness of the computer [5]. This includes providing researchers with powerful platforms that can be used to experiment with different methods of Artificial Intelligence (AI). One practical application of this has been RoboCup, a competition to design robots that play soccer, most of which use AI techniques already being developed in computer games [16]. Many commercial off-the-shelf games are now finding a practical use within the field of military simulation, for example Delta Force 2 and Steel Beasts [8]. This has been reinforced with the U.S. Army investing millions of dollars to work with developers to create various games for simulation purposes.

In computer games, the term ‘game tree’ is used to describe a directed graph whose nodes are positions in a game and whose edges represent moves. The problem is to find an optimal move in a game tree for a certain turn. Each leaf of the game tree is a possible game outcome. In a tree-search based algorithm, as much of the game tree is traversed as possible to find an optimal move. Game tree complexity is the total number of games that can be played or how many leaves the game tree has. A high complexity game has more leaves in the tree than a low complexity game.

Monte Carlo Simulation is a problem solving technique that creates random trials to approximate a solution to a problem. We use the Monte Carlo Method to generate games using only random moves. The final outcome of the game is then evaluated and after many games have been played the Monte Carlo program searches for moves that have been calculated to have high win rates. This simple implementation can be improved by using other techniques such as UCT (Upper

Confidence Bounds applied to Trees) [10]. This technique improves a Monte Carlo algorithm by starting the game with a tree search of possible good moves followed by a random simulation.

In the game of Go, a Monte Carlo approach first appeared in 1993 [3] and a recent Monte Carlo Go program has been very successful [10]. There has been a lot of work put into Monte Carlo Go, primarily because the high degree of combinatorial complexity has made the conventional method of using a tree-search and an evaluation function less effective. This is because the high complexity means a tree-search based technique can only look ahead a few turns, allowing little long term strategy.

This project tests how successful a Monte Carlo technique can be at the game Othello. It also looks at ways of improving the basic Monte Carlo algorithm. These include the effects of increasing the number of simulations, utilising a heuristic function to bias the exploring of moves and the importance of an opponent model.

In Chapter 2 the game Othello is introduced. We explore the history of the game and what makes this an interesting game to study. We also look at the types of strategies involved with playing well and some past successful players. In Chapter 3 we explore the Monte Carlo technique and ways to improve its performance. Chapter 4 summarises the algorithm we implemented including our additions from the basic Monte Carlo. Chapter 5 describes our results. Chapter 6 discusses our findings and possible future work.

CHAPTER 2

Othello

This section describes the history and rules of Othello. The basic strategies for a good Othello player are described as well as features from some successful Othello agents. Lastly the motivation for using Othello as the testbed for testing a Monte Carlo technique is outlined.

2.1 History and Rules

Othello, as it is known today, was introduced in 1975 and credited to Goro Hasegawa, who wrote *How to win at Othello* [17]. However a very similar game called Reversi has been around since 1880. The rules of Othello are simple:

- The game is played on a 8 x 8 board.
- One player is dark and one is white.
- The initial board set up is shown in Fig 2.1.
- Each player takes turns places a disc (their colour up) on the board (dark starts).
- A valid move is one in which a disc is placed on an empty square and outflanks one or more of the opponent's disks between two of mine, which are then flipped over in the process.
- If no valid moves are possible then a player has to pass.
- The game ends when both players must pass (or when the board is filled).
- The aim of the game is to end up with more of one's own colour disks on the board than the opponent.

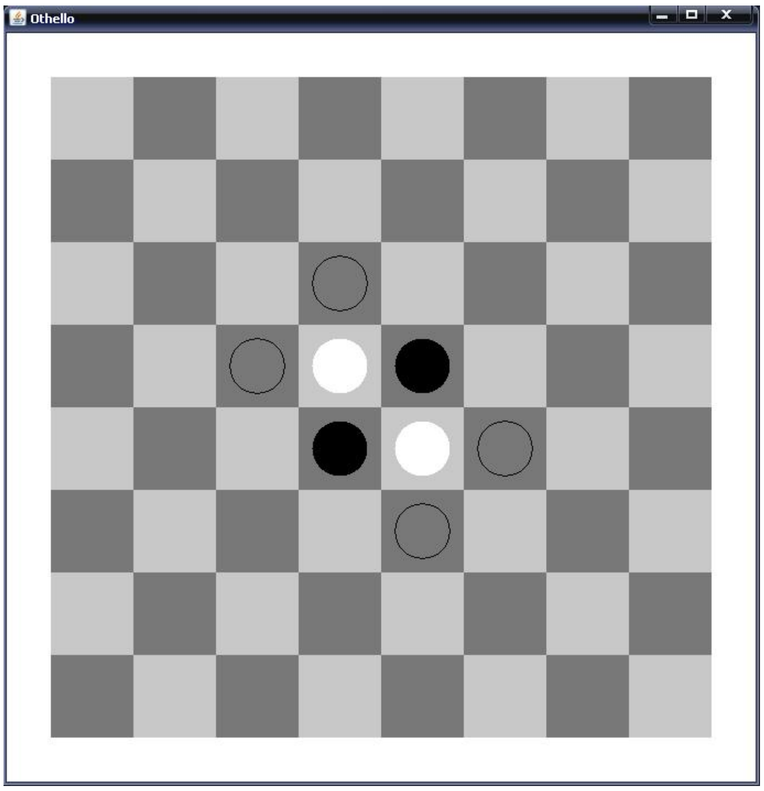

Figure 2.1: My Othello Board.

Figure 2.1 shows the starting Othello board setup and dark's possible moves.

The original Reversi game differed in only two ways. Firstly the first four discs were not preset and had to be played. Secondly each player had a set number of discs and if they ran out, the other player could continue playing the remainder of the game.

In 1980 an Othello tournament was organised for machines to compete against humans. The world champion Hiroshi Inouie won, but lost a game against "The Moor" by Mike Reeve and David Levy. This was the first time in history that a world champion lost a game of skill against a computer. Through the 1980s and 1990s computer Othello players gradually increased in strength. This culminated in 1997, by the 6-0 defeat of the World-champion Takeshi Murakami by Logistello [4]. Today the top computer Othello players are superior to any human players.

2.2 Basic Othello Strategy

The end game strategy for Othello is to maximise the number of one's own colour disks. During the rest of the game however the strategy is very different and involves three main goals.

2.2.1 Stable Disks

A disc that cannot be flipped by the opponent is called a stable disc. These occur when there are no positions that an opponent can ever play that could re-flip them. A disc played in the corner is the most simple example of a stable disc and it is impossible for it to be reflipped. Using the corners as a foundation, it is often easy to create more stable disks along the rows, columns and diagonals alongside corners. Gaining control of the corners and stopping the opponent from gaining them is therefore a very important aspect of Othello. For example if the black player has 23 stable discs then it is assured those discs will remain black's until the end of the game, and black must only have to hold an extra 10 discs to win.

The only way for an opponent to get a corner is if one plays on a square adjacent to the corner. The problem is that when a player has no other moves to play they can be forced to make a bad move. What follows from this is another critical part of Othello strategy called Mobility.

2.2.2 Mobility

In Othello, mobility means maximising the number of possible moves one has while minimising the number of moves the opponent can make. This is beneficial since the fewer moves available to a player the more likely it is that they will be forced to make a bad move, such as described in Section 2.2.1. This is further developed with the notion of frontiers and potential mobility. In Othello a valid move must always be to an empty square that is adjacent to an opponent's disc. This implies that each disc of the opponent's that is adjacent to an empty square could potentially be a valid move for you and vice versa. These discs that are adjacent to an empty square are called frontier discs and make a set of discs called the frontier. The larger the opponent's frontier the more possible moves one has. Potential mobility is a part of mobility and involves trying to increase the opponent's frontier from minimising yours.

2.2.3 Parity

A closed region of the board is a cluster of empty squares separated from the rest of the board by a line of discs. Parity simply means that a player should try to be the last player to put down a disc in a closed region of the board or the final disc of the game. The player who “flips” last has the advantage that whatever was flipped cannot be flipped again. Therefore a player needs to try to force the opponent to pass if parity is not in their favour. This is closely matched with mobility since the player that has a higher mobility has greater control of forcing passes and gaining parity.

2.3 Previous Othello Players

In Game AI we use the term “agent” to describe a computer player. The following three Othello agents highlight how evaluation functions have evolved [4]. All three use a variation of the minimax algorithm described in Section 3.1.

2.3.1 Iago (1982)

This agent uses a classic, hand-crafted evaluation function that combines the elements:

Edge Stability uses a pre-calculated table to determine a measure of stability for the discs lying on an edge.

Internal Stability computes the number of stable discs by using an iterative algorithm.

Current Mobility as described in Section 2.2.2.

Potential Mobility as described in Section 2.2.2.

This agent uses simple strategy techniques described in Section 2.2 without using much pre-computed data.

2.3.2 Bill (1990)

This agent is partly pattern-based and feature weights for each elements of the evaluation function are learned. The evaluation function combines the various elements:

Edge Stability is the same as for Iago.

Current Mobility uses a more complex evaluation that includes different weightings for certain moves.

Potential Mobility is the same as for Iago

Sequence Penalty looks for long sequences of equal colour discs and assigns either a positive or negative weighting depending on the location on the board.

The major difference between Bill and Iago is the increased use of pre computed data. This speeds up all four features of the evaluation function. The more complex evaluations and faster look ups have allowed Bill to beat Iago 100% of the time while only using 20% of the time needed by Iago.

2.3.3 Logistello (1994)

This is the program that beat the world champion, Takeshi Murakami, in 1997. This program relies entirely on pre computed tables. The values in the tables are also calculated/learned by using sample data instead of parameter guessing. Complex patterns are used to evaluate the board from the tables. This makes Logistello an order of magnitude faster than Bill and Iago, which rely on calculating certain features during a move.

2.4 Roxanne

Roxanne is a very simple agent created by Roxanne Canosa [6]. Roxanne picks a move in this order:

1. A corner move.
2. In the center 4 x 4 area.
3. Along an edge, but not next to a corner.
4. Inner edges, but not diagonal from the corner.
5. Squares surrounding a corner.
6. Pass.

This ordering is summarised in Table 2.1, which shows the board and the priority given to each position.

1	5	3	3	3	3	5	1
5	5	4	4	4	4	5	5
3	4	2	2	2	2	4	3
3	4	2			2	4	3
3	4	2			2	4	3
3	4	2	2	2	2	4	3
5	5	4	4	4	4	5	5
1	5	3	3	3	3	5	1

Table 2.1: The priorities Roxanne gives to each position on the board.

Roxanne is created from combining many of the strategy elements from Section 2.2. Most importantly corner moves have the highest priority. Therefore if Roxanne can make a corner move it will. This is a very simple yet critical strategy, since the corners play such a crucial role in the outcome of an Othello game. The positions directly adjacent to the corners are given the lowest priority, to hinder the opponent's chances of gaining the corners. Mobility also plays a factor in Roxanne's moves. Playing central moves early on in the game can help increase one's own mobility and limit that of the opponent by increasing the number of internal discs, those that do not touch any empty squares, and therefore reducing the frontier.

2.5 Motivation for using Othello

Othello is chosen since it is easy to implement but allows for enough strategy to make it interesting to study. There is also an extensive literature on the game and the computer algorithms available. This is useful as it allows the program to be compared with other programs and it can be judged on how strongly it performs against well established algorithms. Othello also has an average computational complexity (defined in Section 3.1). It is not as high as games like Chess or Go but higher than Checkers or Connect Four.

CHAPTER 3

Monte Carlo

A Monte Carlo (MC) algorithm involves making numerous random trials in order to approximate a solution. This approach is used in many fields. In Mathematics the MC method is often used when numerical solutions are too complicated to solve analytically, for example multidimensional definite integrals [14]. In computer science Monte Carlo has been used in global illumination algorithms for computer graphics [7]. This algorithm is stochastic since it uses random numbers. This chapter will discuss the basics of game trees and the minimax algorithm. Then will explore the basic MC algorithm and move on to improvements which can be made to improve performance.

3.1 Game Trees and Minimax

The term 'game tree' refers to a directed graph whose nodes are positions in a game and whose edges represent moves. The problem is to find an optimal move in a game tree for a certain turn. Each level of the tree is called a ply and each leaf of the game tree is a possible game outcome. In a tree-search based algorithm, as much of the game tree is traversed as possible to find an optimal move. For example a 2-ply search generates the game tree for all of one's possible moves and the opponent's possible moves in return. Game tree complexity is approximated by the total number of games that can be played or how many leaves the game tree has. A high complexity game has more leaves in the tree than a low complexity game.

The Minimax algorithm attempts to find the best possible move to play. Given an infinite amount of time perfect play is guaranteed since the whole game tree can be searched. However with limited time a Minimax algorithm must use an evaluation function to approximate the strength of a certain board position throughout the game. This way a minimax algorithm can cut off its search at a set number of plys and use the evaluation function to estimate how good it would be for a player to reach that position. The choice and quality of its evaluation function is therefore critical to the performance of the minimax algorithm.

3.2 Basic MC algorithm

For computer games we can use MC to choose the best move to play on a particular turn. For each possible move on that turn we can run MC simulations on each move to approximate how good that move is. A simulation involves:

1. Pick a certain move.
2. Make random moves until the game is finished.
3. Calculate the winner and report back the result up the tree to the initial move.

Using many of these simulations, a simple MC algorithm would be to:

1. Find out all possible moves to make.
2. Divide the total number of simulations by how many possible moves there are.
3. Run random simulations this many times for each move.
4. Find the move that had the highest chance of winning. From this, infer that such a move is the best in this situation.

This algorithm has numerous advantages over traditional tree search based algorithms. These include:

- This algorithm requires no domain knowledge. This is very impressive. Tree search based algorithms (for example the minimax algorithm described in Section 3.1) rely heavily on the use of evaluation functions. These involve evaluating a board on how good the board is for each player. The problem with this is that the programmer needs to know a large amount of domain knowledge to be able to program such a function with even a small amount of accuracy.
- A consequence of needing no domain knowledge means that an MC algorithm is highly portable between games. In fact if the game interfaces are set up the same there should not be any need to change a MC agent.

- An MC algorithm is very easy to program.

Now we will look at ways to improve the basic MC algorithm. One problem with the basic MC algorithm is that all moves get the same number of simulations. In many games often some moves are far worse than other choices (e.g. next to a corner in Othello). MC will keep trying that move for its allotted number of simulations even if it has already been found to be much worse. On the other hand, two “seemingly” good moves would benefit greatly from having extra simulations to help differentiate which is better. What this boils down to is a tradeoff between exploitation and exploration. Do we continue checking what we know gives an adequate outcome or try something new that may be better?

A well studied example of the exploitation vs exploration tradeoff is the multi-armed bandit problem. This is a simple machine learning problem. A gambler sits at a slot machine with multiple levers. Each lever has an associated expected payoff that is initially unknown to the gambler. The gambler’s aim is to maximise the returns through iterative pulls. Hence, there is a tradeoff between exploitation of the current best lever and exploration in case another lever is actually better. Here we can see the direct relation to our problem of running simulations on the moves that seem best but checking whether other moves are actually better. The Section 3.3 describes the algorithm, Upper Confidence Bounds (UCB1), which is a solution to the multi-armed bandit problem and has been proven to explore the best option exponentially more than any other [1].

Using the UCB1 algorithm means we can run simulations of the more effective initial moves more often. This is a good start, but we can extend this idea further. What if there is a really bad move on the second ply of the game tree? We then have the exact same problem as above. Therefore, what we can do is extend the idea of UCB1 to work throughout the game tree. There is an algorithm called UCT (Upper Confidence Bounds applied to Trees) that does precisely this [12]. It treats each node of the search tree as an independent multi-armed bandit problem. This algorithm uses the UCB1 algorithm to traverse down the search tree by applying it to each node to select the next branch to follow. This is discussed in more detail in Section 4.2.

The basic MC algorithm simply chooses a random move every turn. This is prone to inaccurate results since the random simulation can choose moves that are bad and a real opponent would never play them. Domain knowledge can be added to the random simulations to help make them more accurate [10]. This way we can bias the simulations to play more likely moves. Our aim is then to apply an heuristic to the random move chooser such that the random simulations are closer to a board’s real value. Two parameters have to be balanced for a strong heuristic function. The first is that the heuristic should provide more accurate results and secondly the heuristic function should be fast to evaluate. This will be discussed in more detail in Section 4.3

3.3 UCB1

The K-armed bandit problem is the problem described above with K arms. The problem is defined with random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, where each i is the index of different arms of the gambling machine. Successive plays of machine i give the returns $X_{i,1}, X_{i,2}, \ldots$. These values are independent and identically distributed according to a certain but unknown law with expectation μ_i. Independence also holds for different levers, i.e., $X_{i,s}$ and $X_{j,t}$ are independent for each $1 \leq i < j \leq K$ and each $s, t \geq 1$. Our goal is to now build an allocation policy for mapping the next arm to be played based on the past selections and returns given. Let $T_i(n)$ be the number of times lever i has been selected and played after the first n plays. We can now look at the expected loss E. Regret is the difference between Expected Loss and Actual loss. The regret after n plays is defined by :

$$\mu^* n - \sum_{j=1}^{K} \mu_j E[T_j(n)] \text{ where } \mu^* = \max_{1 \leq i \leq K} \mu^* \tag{3.1}$$

We then use an algorithm UCB1 [1] which ensures that the optimal lever is played exponentially more often than any other lever uniformly when the rewards are in [0,1]. We use:

$$\bar{X}_{i,s} = \frac{1}{s} \sum_{j=1}^{s} X_{i,j} \ , \ \bar{X}_i = \bar{X}_{i,T_i(n)}, \tag{3.2}$$

This leads to:

UCB1

- Initialization: Play each machine once.
- Loop: Play machine j that maximises $\bar{X}_j + \sqrt{\frac{2 \log n}{T_j(n)}}$, where n is the overall number of plays done so far.

The first term $\bar{X}_j$ (T_1) is the average value so far. This naturally increases as an MC simulation reports a win game. The second term $\sqrt{\frac{2 \log n}{T_j(n)}}$ (T_2) is a function that increases when the number of times this node has been looked at is low. Therefore the final function balances T_1 and T_2 to strike the balance between exploitation and exploration. For example say we have found one very good move and one not so good move. The good move will keep being chosen (due to T_1 being high) until T_2 of the other move increases enough to make up for it. At this point the second move will be played. This lowers its T_2 but gives it an opportunity to improve its T_1 if the MC simulation reports back a win.

CHAPTER 4

Proposed Approach

This section describes the goals of the project and the implementation details that allowed us to accomplish these goals.

4.1 Project Aims

This project aims to address

1. How well does an MC algorithm perform at Othello.
2. What improvements can be made to the standard MC algorithm.
3. How well do these improvements compare against each other.

More specifically we want to know what the effects are of using:

- a higher number of MC simulations
- heuristic functions
- opponent modelling
- two heuristic function
- noise in the opponent model

4.2 Implementing UCT

The UCT algorithm starts by adding the root node to the search tree and its children. The value of each child is initially set to a large number so that UCB1 always chooses a child that hasn't been chosen yet (Figure 4.1).

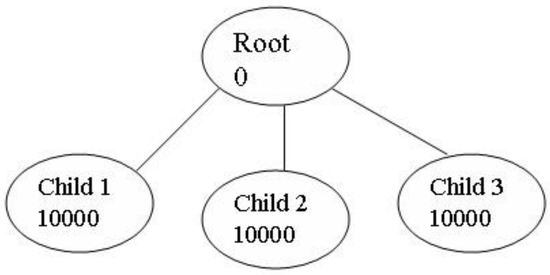

Figure 4.1: UCT. Root node and children.

Now we start at the root and use UCB1 to choose which child to descend to. Since none have been chosen yet it will choose one at random. Let's say it selects child 1. Now we try to descend further but find that we are at a leaf node. We now add this node's children to the tree and run a Monte Carlo simulation as shown in Figure 4.2.

This continues in this manner until all three children have been tried and we arrive in the state shown in Figure 4.3.

This process can continue for as many runs as is allowed or resources permit. A benefit of this approach over a Minimax algorithm is that after many simulations the tree is created asymmetrically and naturally explores the moves that are shown to be more promising, whereas a Minimax tree is created at a uniform depth across all moves. Another positive is that if the UCT algorithm is stopped prematurely the algorithm will have a good estimate for the best move. However if a Minimax algorithm is stopped prematurely the tree will be only be partly formed and may not have even looked at many move possibities.

This UCT algorithm is given in pseudo code in Algorithms 1, 2 and 3.

Algorithm 1 playOneSequence(rootNode)

```
1: node[0] := rootNode; i = 0;
2: while node[i] is not leaf do
3:     node[i+1] := descendByUCB1(node[i]);
4:     i := i + 1;
5: end while
6: updateValue(node, -node[i].value);
```

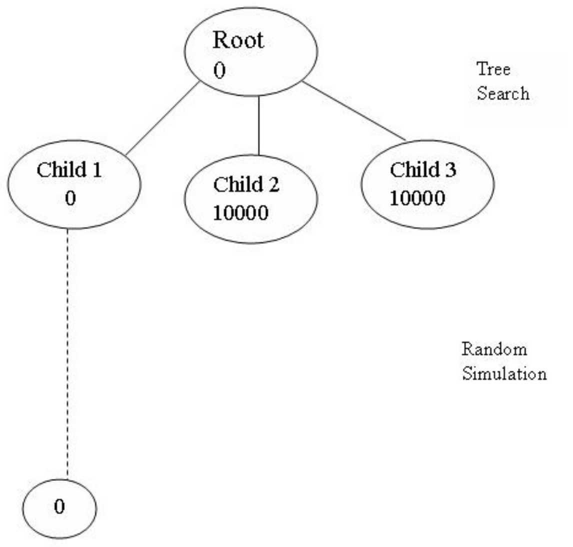

Figure 4.2: UCT. Root node and children after one random simulation.

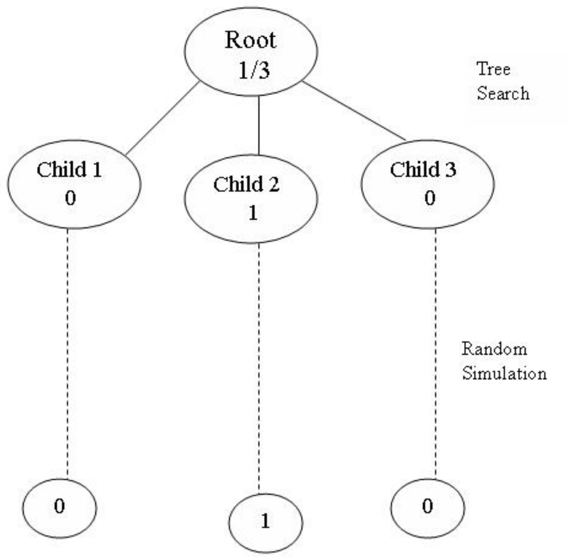

Figure 4.3: UCT. Root node and children after three random simulations.

Algorithm 2 descendByUCB1(node)

```
1: nb := 0;
2: for i := 0 to node.childNode.size() - 1 do
3:     nb := nb + node.childNode[i].nb;
4: end for
5: for i := 0 to node.childNode.size() - 1 do
6:     if node.childNode[i].nb = 0 then
7:         v[i] := 10000;
8:     else
9:         v[i] := $\frac{-node.childNode[i].value}{node.childNode[i].nb} + \sqrt{\frac{2\log nb}{node.childNode[i].nb}}$
10:    end if
11: end for
12: index := argmax(v[j]);
13: return node.childNode[index];
```

Algorithm 3 updateValue(node, value)

```
1: for i := node.size() - 2 to 0 do
2:     node.value := node[i].value + value;
3:     node[i].nb := node[i].nb + 1;
4:     value := -value;
5: end for
```

4.3 Heuristic Functions

The problem with using random moves is that each simulation is very inaccurate as very bad moves that would never normally be played are being considered. During a MC simulation a random game is played till the end. More precisely, the MC player plays a virtual game with itself and makes a random move for both its own turn and for the opponent's. Instead of making random moves the simulation can choose a move to play by using a certain criteria. This is called a heuristic function. For example a heuristic function could be to always play the move that flips the most discs. When implementing a single heuristic function, both the MC player's turn and the opponent's turn are chosen by the heuristic function. So the heuristic is played from black's perspective and then white's perspective alternatively until the end of the game. An MC agent using heuristic function h in this manner is written as MC(h).

This is implemented by creating each heuristic function as a new agent. This allows the heuristic functions to play independently of an MC player and hence compare the strengths of each heuristic function as an agent on its own. Then when running a simulation in an MC agent we can simply call the heuristic function just as if it were actually playing a game. This comes at a price since choosing a random move will always be the fastest way of choosing a move. So if a heuristic function carries out complex calculations to increase its accuracy, then it is going to be slower. Hence a balance must be achieved between a strong heuristic function and a fast one.

4.3.1 Roxanne

The Roxanne agent is described in Section 2.4. Instead of choosing a random move, we can use Roxanne as the move choosing heuristic. A Roxanne agent performs much better than a Random Agent, and therefore we hope that using Roxanne as a heuristic will be more accurate than randomly choosing. An advantage for Roxanne is that the speed of evaluating is almost as fast as choosing randomly. This is because Roxanne simply bases the move choice on the board positions and is only a zero-ply look ahead search.

4.3.2 Mobility

In Section 2.3, mobility was described as an important feature in evaluation functions for minimax algorithms. Therefore mobility may also be a suitable heuristic function in our MC agent. The mobility of a board position is calculated as the number of the agent's possible moves subtract the number of the opponent's possible moves. A mobility agent then must calculate the mobility for each of the agent's possible moves and find the one with the largest. Unfortunately this is a one-ply look ahead search since we need to know the state of the board after making each possible move. This makes a mobility agent much slower than either Random or Roxanne.

4.3.3 Pot.Mob.

This agent uses potential mobility as its measure or strength. Its implementation is the same as for the mobility agent except that it uses potential mobility instead of mobility. This is calculated by counting the number of opponent pieces that are adjacent to an empty square (the frontier) and subtracting the agent's frontier. This agent also suffers from an even slower speed than mobility.

4.3.4 CPM

We can now combine different parts of evaluation functions into a single heuristic function. The most important feature used is the corner ratio. The corner ratio is defined as the number of corners held by the agent minus the number of corners held by the opponent. This measure is given the highest priority and simply means that if the agent can capture a corner it will. Next the two measures mobility and potential mobility are combined together with equal weightings. This value is then used as a secondary measure to differentiate between moves that have an equal corner ratio.

4.3.5 CPMR

This is the final agent and combines all the previous agents. The CPM agent will often arrive at a draw between moves after calculating each move's value. In this case CPM simply returns a random move out of these moves. CPMR however uses the extra measure of Roxanne to separate these moves further. This has the benefit of more control over the moves chosen by CPMR but means CPMR is becoming more deterministic.

4.4 Opponent Modelling

The reasoning for using a heuristic function is to increase the accuracy of each simulation by providing a result similar to the actual value of a move. The actual value of a move depends on the opponent. For example a strategy that works against Roxanne may be very different to a strategy that works against Mobility. From this we can see that the heuristic does not necessarily have to be a strong player but instead can accurately model the opponent. In the extreme case that we have a perfect opponent model, then we always know what move the opponent will play at any board state. Thus if this perfect model is used as a heuristic function the results returned after the simulations are no longer probabalistic approximations, but rather they are the exact values for each path of moves. This is obviously very good since very few simulations would be needed to find the optimal move to play.

If the opponent model is perfect then a MC approach would not be necessary since the game outcome could be calculated before the game even begins. However we are interested in the case where an opponent model can be approximated. This can be simulated by the agents MC(*heuristic*) playing against *heuristic*. For example in the match up between MC(Roxanne) and Roxanne, the MC(Roxanne) agent is using a very good approximation to a perfect opponent model of Roxanne as a heuristic function (non-determinism makes the model less than perfect). We hypothesise that using a more accurate opponent model is more important than using an otherwise stronger heuristic function.

4.4.1 Combining Heuristic Functions

One problem with using a single heuristic is that the heuristic is being used for both the opponent's turn and the agent's turn when performing a random simulation. In the example in Section 4.4 Roxanne is being used as the heuristic for both turns throughout MC(Roxanne)'s simulations. The opponent modelling is only necessary when playing the opponent's move. An improvement is to use two different heuristic functions, one for the opponent's move and one for your own move. This is written as MC($h1$, $h2$) where $h1$ is the heuristic for the opponent's moves and $h2$ is the heuristic used for the MC agent's moves. This combines the benefits of opponent modelling while still partly using a (possibly) stronger heuristic function.

4.4.2 Implementing Noise in the Opponent Model

If the random heuristic is bad and a perfect opponent model is good, this leaves the question of whether an opponent model that is partly accurate is any good. Is it worth using a model that is 90%, 70% or 50% accurate? This can be simulated in an experiment by creating noise within the opponent model. In the agent MC($h1$, $h2$), $h1$ models the opponent and is therefore forced to make a random move with a probability of ϵ.

CHAPTER 5

Experiments and Results

This chapter describes the experiments we performed to test the agents and the results found.

5.1 Environment

All experiments were run on a linux server. For each experiment between two players they were run for 100 games (50 white, 50 black) repeated 10 times (for a total of 1000 games each comparison). These ten samples were then averaged and the average number of wins used for comparison. This number will therefore be close to 100 for an extremely good agent, around 50 for similar agents and less than 50 for a not so good agent. The number of wins is for agents listed on the left and to be read horizontally. The same notation from Section 4.3 is used where an MC agent that uses heuristic function h is written as MC(h). If two different heuristic functions are used as described in Section 4.4.1, it is written as MC($h1$, $h2$) where $h1$ is used for the opponent's moves and $h2$ is used for one's own moves during each random simulation.

5.2 AgentX vs. AgentY

Firstly the non MC agents are compared against each other. This consists of Random, Roxanne, Mob, Pot.Mob, CPM and CPMR as simple agents. These results are listed in Table 5.1.

From table 5.1 it is clear that Roxanne is by far the strongest agent by winning 81.7% of all games and Random is by far the weakest with only 21.46% wins. The other agents are ordered (descending strength) CPMR, CPM, Mob and Pot.Mob with 57.28%, 51.2%, 41.98% and 41.74% win rate respectively. Random's loss is be be expected as it is not using any game knowledge. However Roxanne is

	Random	Roxanne	Mob	Pot.Mob	CPM	CPMR	Avg
Random		15.8	25.3	35.7	16.1	14.4	21.46
Roxanne	81.8		80.7	90.2	77.9	77.9	81.7
Mob	72.8	16.7		57.5	41	21.9	41.98
Pot.Mob.	61.6	8.9	40.2		44.4	53.6	41.74
CPM	81.9	20.5	58.1	55.6		39.9	51.2
CPMR	84	21.1	76.9	46.4	58		57.28

Table 5.1: Agents vs. Agents

surprisingly good considering it is using a 0-ply look ahead compared with the other agents using a 1-ply look ahead.

5.3 MC(AgentX) vs. AgentY

This is a comparison between how well the agents perform against MC(h) where h is one of the agents. This gives an initial idea of how strong an agent is when used as a heuristic function. All the MC(h) agents are using 50 simulations per turn. These data are shown in Table 5.2.

	Agents					
	Random	Roxanne	Mob	Pot.Mob	CPM	CPMR
MC(Random)	97.7	74.8	71.4	74.2	55.2	49.9
MC(Roxanne)	98.5	84.7	73.5	78	63.7	58.8
MC(Mob)	98	79.6	92.4	88	71.6	69.8
MC(Pot.Mob)	96.7	78.2	86.9	95	75.6	73.3
MC(CPM)	99.3	84.7	91.3	95.8	94.8	94.4
MC(CPMR)	99.4	85.8	91.7	96.3	94.7	95.6

Table 5.2: MC(h) at 50 Simulations vs. each Agent

The first observation from Table 5.2 is that all the MC(h) agents perform very well. With the exception of one match up (MC(Random) vs. CPMR) all the MC(h) agents outperform the non-MC agents. The CPM agent is a strong agent from a strategic point of view. It has a very large amount of domain knowledge , as described in Section 4.3.4. Yet an MC agent using zero domain knowledge can win 55.2% of the time when using only 50 simulations (this number can be set as high as a few thousand). Also by using MC(CPM) it can win 94.8% of the time against CPM. Considering these two agents have the exact same amount of domain knowledge, this is a huge figure. These are very promising results for the MC technique.

Opponent Modelling is where the MC agent uses a model of its opponent as a heuristic function, explained in more detail in Section 4.4. By looking along the diagonal in Table 5.2 (top right, to bottom left), these are the values that correspond to an MC(h) agent using a model of its opponent, h, (for example MC(Roxanne) vs. Roxanne). These values are clearly abnormally high. By ignoring the Random agent we find that MC(Roxanne), MC(Mob) and MC(Pot.Mob) all are strongest when matched against their respective agent. Also MC(CPM) and MC(CPMR) both perform best against Pot.Mob which is still strongly connected with opponent modelling since CPM and CPMR both use potential mobility as a key part of their heuristic function. These results show a clear positive relationship between opponent modelling and playing strength.

Due to this opponent modelling feature, it makes using Table 5.2 as a measure of the strength of MC(h) difficult since it is hard to tell which values are from pure agent strength and which are from opponent modelling. Even so, CPM and CPMR appear to make very strong heuristic functions since MC(CPM) and MC(CPMR) are the best two agents against Pot.Mob, CPM and CPMR. They are very close to MC(Mob) against Mob and in fact do slightly better or equal to MC(Roxanne) against Roxanne. Since MC(Roxanne) has the added advantage of opponent modelling this shows that CPM and CPMR are simply very strong as heuristic functions.

From Table 5.1 we found Roxanne to be the strongest agent against other agents, by a substantial margin. However by looking at each agent played against MC(Random) we find CPMR to be the strongest agent and is in fact the only agent to not be beaten by every MC agent (losing 49.9% of all games, indicating rough equivalence in playing strength to MC(Random)). This indicates that agent vs. agent strength is not directly related to agent vs. MC(h) strength.

5.4 MC(AgentX, CPMR) vs. AgentX

This is a comparison of how well an MC(h, CPMR) agent does against an agent h. In this scenario we use a combination of heuristic functions (from Section 4.4.1) such that h is being used to model the opponent's moves and CPMR is being used for the MC agent's moves during the MC simulations. This is different to the experiments above since MC(h) uses the heuristic h for both sides' moves. Table 5.3 has the row containg these new results appended to Table 5.2 to allow for direct comparisons.

	Agents					
	Random	Roxanne	Mob	Pot.Mob	CPM	CPMR
MC(Random)	97.7	74.8	71.4	74.2	55.2	49.9
MC(Roxanne)	98.5	84.7	73.5	78	63.7	58.8
MC(Mob)	98	79.6	92.4	88	71.6	69.8
MC(Pot.Mob)	96.7	78.2	86.9	95	75.6	73.3
MC(CPM)	99.3	84.7	91.3	95.8	94.8	94.4
MC(CPMR)	99.4	85.8	91.7	96.3	94.7	95.6
MC(Agent,CPMR)	97.5	95.2	94.2	96.5	95.4	96.8

Table 5.3: MC(h) at 50 Simulations vs. each Agent with the addition of an MC(Agent, CPMR) vs. Agent row.

From the bottom row of Table 5.3, all the MC(Agent, CPMR) agents perform extremely well (all above 94% wins). Additionally we observe that, with the exception of MC(Random,CPMR), all the new MC(h,CPMR) agents are the strongest agents against each heuristic. The most obvious example is against Roxanne where initially the strongest agent was MC(CPMR) with an 85.8% win rate, whereas with the new results added MC(Roxanne, CPMR) has a 95.2% win rate. Interestingly this is much higher than either MC(Roxanne) or MC(CPMR). This further reinforces the fact that opponent modelling is a key factor in agent performance, but also highlights the benefit of using two heuristic functions.

5.5 MC(AgentX) vs. MC(AgentY)

This experiment is a comparison between each MC($h1$) agent against each other MC($h2$) agent. The goal is to find which agent performs the best as a heuristic function. The number of simulations is set to 50 for all agents. These data are tabulated in Table 5.4.

Table 5.4 shows that MC(CPM) agent at 50 simulations each turn (with 70.22% wins), and CPMR a close second (with 68.04% wins). MC(Random) is by far the worst with only 28.82% wins, with MC(Roxanne) being a dramatic improvement at 42.4%. Interestingly MC(Pot.Mob) just beats MC(Roxanne) but MC(Roxanne) does a little better against MC(CPM) and MC(CPMR) than MC(Pot.Mob) does (27.1% compared with 25.2% and 28.6% against 25.8% respectively).

	MC Random	MC Roxanne	MC Mob	MC Pot.Mob	MC CPM	MC CPMR	Avg
MC(Random)		36	32.9	36.2	19.2	19.8	28.82
MC(Roxanne)	63.8		45.4	47.1	27.1	28.6	42.4
MC(Mob)	67	54		52.8	29.2	32	47
MC(Pot.Mob)	63.6	52.7	47.2		25.2	25.8	42.9
MC(CPM)	80.5	72.8	70.8	74.6		52.4	70.22
MC(CPMR)	79.7	71.3	68	74	47.2		68.04

Table 5.4: MC($h1$) vs. MC($h2$) at 50 Simulations.

From Section 5.2 we found the strength of the agents against other agents to be ordered as:

1. Roxanne
2. CPMR
3. CPM
4. Mob and Pot.Mob
5. Random

From Section 5.3 we found the strength of the agents against MC(h) to be ordered as:

1. CPMR
2. CPM
3. Mob
4. Pot.Mob and Roxanne
5. Random

From Table 5.4 the strength ordering for MC agents using each heuristic function is:

1. CPM

2. CPMR
3. Mob
4. Pot.Mob and Roxanne
5. Random

By comparing the final two orderings we see that the only major difference is a swap of CPM and CPMR. What this means is that the strength of an MC(h) agent is directly proportional to to the strength of h as an agent against an MC agent. However when comparing the first ordering Roxanne is the strongest agent yet it is only an equal fourth in the final two orderings. This means that agent vs. agent strength is not such a good measure of heuristic function strength.

5.6 Equal Time Allowed for Each Move

The problem with the above results is that some heuristic functions take much longer to calculate than others. For example the Random heuristic can perform 10 times as fast as CPMR and therefore could perform 10 times as many MC simulations in the same time period. To make comparisons fair, the time per move is limited rather than a fixed number of simulations. This means that if a Monte Carlo agent uses a heuristic that takes longer to calculate, it will not be able to run it for as many MC simulations. The above two experiments are repeated under this new limitation in Table 5.5, Table 5.6 and Table 5.7.

	MC Random 500	MC Roxanne 485	MC Mob 85	MC Pot.Mob 50	MC CPM 50	MC CPMR 50	Avg
MC(Random)		18.8	67.6	79.5	58.1	56	56
MC(Roxanne)	81.1		87.5	92.1	84	79.2	84.78
MC(Mob)	32.3	12.5		64.1	38.6	39.8	37.46
MC(Pot.Mob)	20.4	7.9	35.9		25.2	25.8	23.04
MC(CPM)	41.9	15.9	61.2	74.6		52.4	49.2
MC(CPMR)	43.7	20.7	60	74	47.2		49.12

Table 5.5: MC($h1$) vs. MC($h2$) with equal time per move.

Table 5.5 contains large differences to the results found in Table 5.4. Roxanne is shown to be the strongest heuristic function by far with 84.78% wins. Interestingly, the next strongest is Random with 56% wins. This shows the importance of the number of simulations since MC(Random) and MC(Roxanne) used 500 and 485 simulations each respectively, whereas MC(Mob) used 85 and the rest only used 50. MC(Pot.Mob) is the worst performer since it is as slow as MC(CPM) and MC(CPMR) but only as strong as Roxanne when using equal simulations. Mob is a little bit faster, thus allowing 85 simulations but that is still not enough to beat CPM or CPMR.

The comparison of Table 5.5 and Table 5.4 highlights two major revelations. Firstly, MC(Roxanne) jumped from being the second weakest agent (when using the set 50 MC simulations) to being the best agent when using a variable number of MC simulations. By comparing two single results we find that MC(Roxanne) won only 27.1% against MC(CPM) originally and then wins an astounding 84% with the extra MC simulations. This clearly shows the dramatic effect the number of simulations can have on the strength of a MC agent. Secondly, MC(Random) wins 56% against MC(CPMR). Considering that MC(Random) is using ten times as many MC simulations as MC(CPM) this is a fairly close result. This shows that even when using far less MC simulations, an agent using a strong heuristic function can perform well. Hence, these two points highlight an important trade-off between MC simulations and heuristic strength.

Table 5.6 is the same as Table 5.5 except this time the number of simulations has been doubled for every agent. The ordering of the strength of each agent has not changed. But we can see that MC(Random) is doing worse than in Table 5.5. In the first table MC(Random) wins 58.1% against MC(CPM), whereas in the second table MC(Random) and MC(CPM) are equivalent (the difference is not statistically significant). Therefore, we find that more MC simulations are beneificial but only up to a point. On the flip side, a strong heuristic is beneficial but not at the expense of too many MC simulations. This idea is further explored in Section 5.7.

	MC Random 1000	MC Roxanne 970	MC Mob 170	MC Pot.Mob 100	MC CPM 100	MC CPMR 100	Avg
MC(Random)		15	60.8	77.8	50.1	55.5	51.84
MC(Roxanne)	84.8		84.5	92.5	82.7	85.1	85.92
MC(Mob)	39.1	15.5		70.2	40.9	43.1	41.76
MC(Pot.Mob)	22	7.4	29.6		23.6	23.3	21.18
MC(CPM)	49.9	17.2	59.1	76.2		48.8	50.24
MC(CPMR)	44.4	14.9	56.5	76.5	51		48.66

Table 5.6: MC($h1$) vs. MC($h2$) with equal time per move, but the time doubled.

	Agents					
	Random	Roxanne	Mob	Pot.Mob	CPM	CPMR
MC(Random)	99.8	92	92.2	92.8	85.8	83.2
MC(Roxanne)	100	97.9	94.5	96.1	94.6	92.1
MC(Mob)	99.4	83.9	96.4	92.4	77.1	77.4
MC(Pot.Mob)	96.7	78.2	86.9	95	75.6	73.3
MC(CPM)	99.3	84.7	91.3	95.8	94.8	94.4
MC(CPMR)	99.4	85.8	91.7	96.3	94.7	95.6

Table 5.7: MC(h) with equal time per move vs. each Agent

Table 5.7 further supports the importance of the opponent model from Section 5.3. Tables 5.5 and 5.6 established that MC(Roxanne) is the strongest player under time limitations. However against CPMR, MC(Roxanne) wins 92.1% whereas MC(CPMR) wins 95.6%. This is an amazing result considering MC(Roxanne) is using almost ten times the number of simulations as MC(CPMR). Due to opponent modelling the MC simulations run using CPMR as a heuristic function must be ten times more accurate then using Roxanne.

5.7 Comparison of Two MC Agents at Varying Times per Move

Table 5.5 uses a set number of simulations based on them using a set amount of time per move. Table 5.6 is the same except that all the agents use twice the number of simulations. Now we extend this idea further by running comparisons between two agents and four and eight times the base number of simulations. Table 5.9 compares this data between MC(Roxanne) and MC(CPMR). If the relationship between the strength of an agent and the number of simulations were linear we would expect that as long as two agents maintain the same factor between their number of MC simulations their strengths should remain the same relative to each other. We can see that CPM wins only 15.9% when the MC simulations are set at 50, but when the simulations are set at 400 CPM wins 23.1%. This is a substantial difference considering that both times Roxanne was using the same multiple of CPM's number of MC simulations.

When comparing the same results but using MC(Random) in Table 5.8 the results are even more pronounced where the win rate drops from 58.1% down to 29.9%. What this points to is that the relationship between strength and simulations is not linear but instead non-linear as shown in Fig 5.1. At the smallest number of simulations Roxanne might be at the upper end of the B section, whereas CPM is sitting at the lower end of the B section. As the number of simulations is multiplied to higher amounts, we find that Roxanne is experiencing a smaller increase in playing strength since it is moving through the C section. CPM on the other hand is travelling through the B section and experiencing a rapid growth in playing strength. Also the difference in results from MC(Roxanne) and MC(Random) indicates that these strength/simulation graphs are different for every agent. The C section for MC(Random) must be much flatter than for MC(Roxanne). Hence MC(Random)'s performance degrades more rapidly as the number of simulations goes up.

	Number of simulations for CPM (Random)			
	50 (500)	100 (1000)	200 (1940)	400 (3880)
MC(Random)	58.1	50.1	43.7	29.9
MC(CPM)	41.9	49.9	56.2	70.1

Table 5.8: MC(Random) v MC(CPM) at different number of simulations

	Number of simulations for CPM (Roxanne)			
	50 (485)	100 (970)	200 (1940)	400 (3880)
MC(Roxanne)	84	82.7	79.7	76.8
MC(CPM)	15.9	17.2	20.3	23.1

Table 5.9: MC(Roxanne) v MC(CPM) at different number of simulations

5.8 The Effect of Noise

This is tied in with opponent modelling. What we want to know is how accurate does an opponent model need to be to remain effective? This was discussed in Section 4.4.2. The noise is added (implemented by making a random move with probability ϵ) to the heuristic function that is used for the opponent's moves. These results are shown in Table 5.10. The results shown are the win percentages for the agents on the left column vs. that particular heuristic function. These results are shown for the amount of noise varying between 0% and 100%.

Table 5.10 shows that the performance of the MC agents decreases as the amount of noise in the opponent model increases. For example MC(CPMR) wins

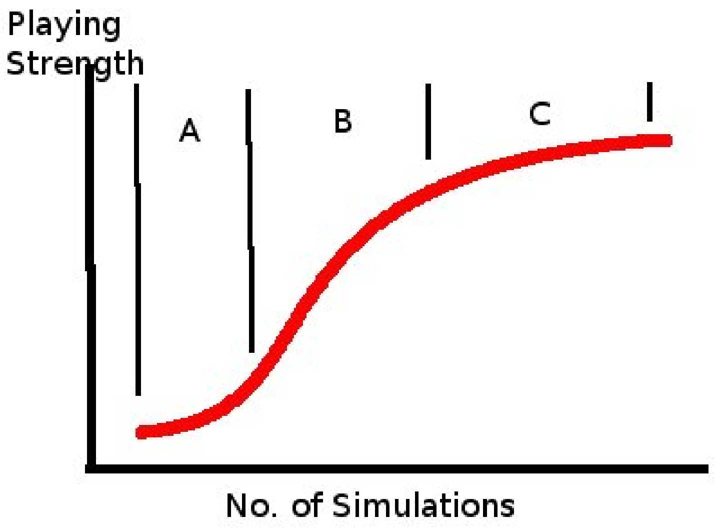

Figure 5.1: Non-Linear relationship between playing strength and the number of simulations

97.4% at 0% noise but only 55.8% at 100% noise. More interestingly the drop in performance from 0% noise to 35% noise (84.2% wins) is less extreme (as one would expect). By plotting these values, in Fig 5.2, we find that for every agent the rate of performance drop is linear (with the exception of MC(Random) which is to be expected). This is a very important result since we now know that even a partly accurate opponent model is still worth doing.

MCA	Noise								
	0%	5%	10%	15%	20%	25%	35%	50%	100%
MC(Random)	96.5	97.3	97.5	97.5	97.2	97.5	97.4	97.2	96.1
MC(Roxanne)	85.1	84.7	86.9	83.3	82.3	82	81.5	79.3	69.4
MC(Mob)	93.7	92.9	91.7	90.4	89.5	89.8	86.5	83.8	70
MC(Pot.Mob)	94.5	93.7	93.3	91.7	90.7	90.2	90	85.7	77.6
MC(CPM)	95.4	94.3	91.3	91.7	89.7	87.1	84.9	80.1	56.3
MC(CPMR)	97.4	93.3	91.2	91.2	87.2	86.3	84.2	75.2	55.8

Table 5.10: MC(AgentX) with noise vs. AgentX

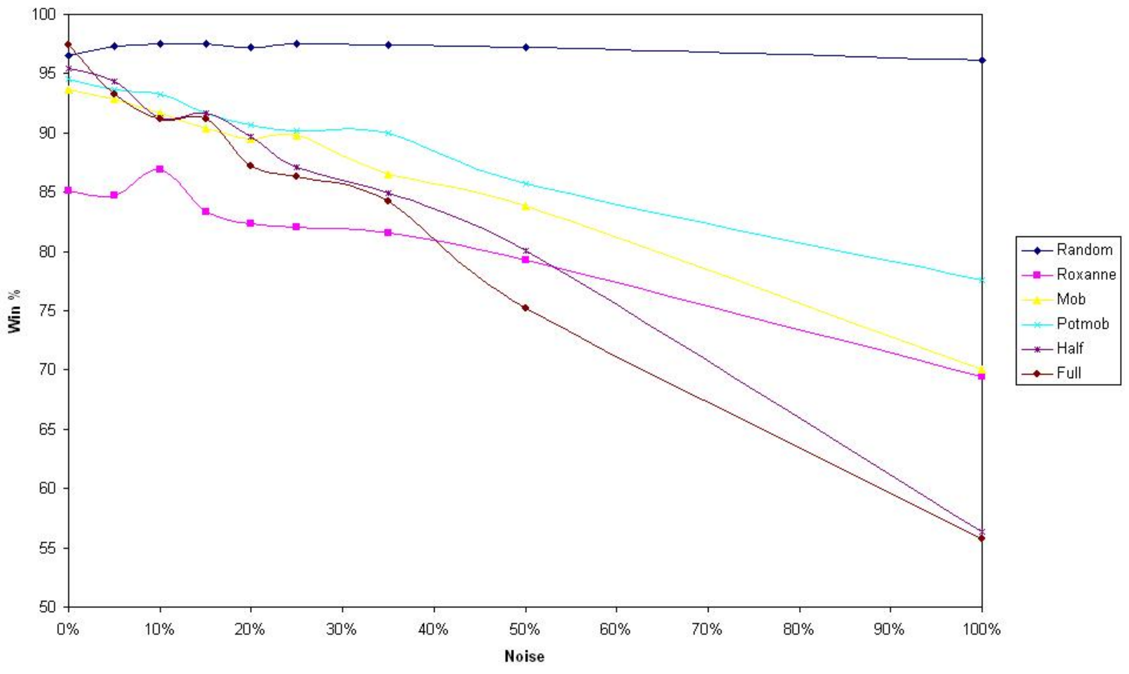

Figure 5.2: Noise

5.9 Monte Carlo Strength Against Humans and Minimax

The Monte Carlo agent was also tested against various human players. This agent used the base UCT algorithm with no heuristic function and was found to be very successful but unfortunately these games were only played against very amateur humans.

Next the MC agent was compared with a basic minimax agent. Various java applets found on the internet were used for testing. All games resulted in a loss for our agent. This was with both a basic MC agent and MC(CPMR) at 1000 simulations. The minimax agents were only searching around 2-4 ply deep. Very shallow by the top Othello computer player standards (Logistello searches 18-23 selective including 10-15 brute-force ply in middle game).

Unless the MC agent can be improved drastically , by factors of 10, our agent has no chance of winning against strong computers.

	Noise					
	0%	5%	10%	15%	20%	25%
MC(Random,CPM)	97.5	98.7	97.8	97.8	97.4	97.7
MC(Roxanne,CPM)	95.2	91.9	94.1	90.9	89.8	90.2
MC(Mob,CPM)	94.2	94.9	93.8	90.4	91.6	89.9
MC(Pot.Mob,CPM)	96.5	95.7	94.9	94	94.1	91.9
MC(CPM,CPM)	95.4	94	92.7	89	88.7	86.5
MC(CPMR,CPM)	96.8	92.5	91.7	89.7	88.4	85.4

Table 5.11: Monte Carlo Agents (CPM) with noise vs Agents

CHAPTER 6

Conclusion

A Monte Carlo algorithm is one that uses repeated random simulations to gather an estimate of the correct value. In Othello a Monte Carlo algorithm plays random games many times to calculate an estimate of the quality of each possible move. For our implementation of Monte Carlo we use an algorithm called UCT which improves this basic Monte Carlo by using a tree search to balance the 'exploitation vs. exploration' problem when choosing which moves to play [12].

This dissertation has examined the advantages and disadvantages or using a Monte Carlo technique in Othello. Traditional tree search based algorithms require high levels of domain knowledge to perform well due to to critical use of evaluation functions [15]. A Monte Carlo approach instead requires zero domain knowledge to perform well. This means that the programmer does not need to be an expert at a game to be able to make a strong agent for it. The next advantage is that due to the lack of domain knowledge needed, a Monte Carlo agent is highly portable between games. If the games are set up with the same interfaces, a Monte Carlo agent for one game will work without modification on any other game. These factors make a Monte Carlo agent very easy to program.

The main disadvantage of a Monte Carlo agent is that at the moment it will not come close to beating a highly specialised agent that utilises a large amount of domain knowledge. For example a strong Othello agent, Logistello, relies heavily on fast board evaluations using patterns [4]. We can add domain knowledge to our Monte Carlo agent to improve performance, but then this takes away many of the advantages described above.

From our experiments we have found that using a strong heuristic function to bias the moves during an MC simulation can greatly improve performance. A Monte Carlo agent that uses the CPM heuristic beats a Monte Carlo agent using no heuristic 80% of the time. However the number of simulations has also be found to be critical. MC(Roxanne) wins 28.6% against MC(CPMR) at 50 simulations each, but when MC(Roxanne) uses 485 simulations this win rate jumps to 79.2%. Therefore a quality heuristic function must balance accuracy with speed of calculation. This is a very important trade-off to balance especially since some agents recieve more benefit than others from increasing the number of simulations.

We have shown the importance of opponent modelling throughout our experiments. Against the CPM agent MC(Roxanne) wins 92.1% of time using 485 simulations, but MC(CPMR) wins 95.6% of games with only 50 simulations. This shows that when the opponent model is accurate we can run it for far less simulations. We have also seen great success with using two heuristic functions. This is where we use one heuristic function to model the opponent's moves and another to model one's own moves. For example at 50 simulations against the agents the best results for all but Random were won by MC(h, CPMR) where h was the agent.

By running experiments between two agents and varying the number of simulations we have found that the strength of an MC agent is not linearly dependent on the number of simulations. Instead the effect of increasing the number of simulations diminishes when we reach higher simulations. This has large consequences since at a low number of simulations MC(Random) is found be be stronger than MC(CPMR) and wins 58.1% of games when using equal time per move. But when the time per move of both agents is increased, and hence the number of simulations of both increases, MC(Random) wins only 29.9% of the time. From this we see that with a low time per move it is best to focus on increasing the speed of a heuristic, but with a high time per move, when the effects plateau, it is best to focus on improving the heuristic function.

The effect of noise ties in with opponent modelling since this is a method of seeing how accurate the opponent model must be to have an effect. What was found is the strength of the agent was linearly dependent on the amount of noise. This is very helpful since we now know that even if the opponent model is only 50% accurate it will still be half as good as a perfect model and far better than having no model.

Future work could involve taking an extremely efficient evaluation function from a very strong Minimax agent and using it as a heuristic function in our MC agent. These evaluation functions are designed to be very fast since they use pattern matching and pre-computed tables. Also it would be very interesting to see the results when the number of simulations is much higher, for example 50,000 or even 100,000. This could be done by making the algorithm work in parallel on multiple computers.

APPENDIX A

Original Honours Proposal

Title:	Comparing a Monte Carlo approach to conventional techniques in game AI
Author:	Ryan Archer
Supervisor:	Luigi Barone

Background

Games

The computer games industry is big business with a study in 2004 showing that 60 percent of all Americans aged six or older play video games [13]. Not only are we interested in games for consumeristic reasons but also for the benefits games and gaming techniques have on society. For example, experts point out that games are an important steppingstone for kids into the world of technology [11]. Students who play computer games tend to be more comfortable with the technology and more adept at using it. Computer games fill the role as being an incubator for many technologies that drive the usefulness of the computer [5]. This includes providing researchers with powerful platforms that can be used to experiment with different methods of AI. One practical application of this has been RoboCup, a competition to design robots that play soccer, most of which use AI techniques already being developed in computer games [16]. Many commercial off-the-shelf games are now finding a practical use within the field of military simulation, e.g. Delta Force 2 and Steel Beasts [8]. This has been reinforced with the U.S. Army investing millions of dollars to work with developers to create various games for simulation purposes.

In computer games, we use the term game tree to describe a directed graph whose nodes are positions in a game and whose edges represent moves. The problem is to find an optimal move in a game tree for a certain turn. Each leaf of the game tree is a possible game outcome. In a tree-search based algorithm, as much of the game tree is traversed as possible to find an optimal move. Game tree complexity is the total number of games that can be played or how many leaves the game tree has. A high complexity game has more leaves in the tree than a low complexity game.

Monte Carlo

Monte Carlo Simulation is a problem solving technique that creates random trials to approximate a solution to a problem. We use the Monte Carlo Method to generate games using only random moves. The final outcome of the game is then evaluated and after many games have been played the Monte Carlo program searches for moves that have been calculated to have high win rates. This simple implementation can be improved by using other techniques such as UCT (Upper Confidence Bounds applied to Trees) [10]. This technique improves a Monte Carlo algorithm by starting the game with a tree search of possible good moves followed by a random simulation.

In the game of Go, a Monte Carlo approach first appeared in 1993 [3] and a recent Monte Carlo Go program has been very successful [10]. There has been a lot of work put into Monte Carlo Go, primarily because the high degree of combinatorial complexity has made the conventional method of using a tree-search and an evaluation function unsuccessful. This is because the high complexity means a tree-search based technique can only look ahead a few turns, allowing little long term strategy.

This leaves us with the question as to whether a Monte Carlo approach may also work well in other games. Does this approach only work well in high complexity games, or will it also work in other types of games too? To answer this question, we must first try and classify games into different types. Games can be roughly divided into two categories

Perfect Information. These are games where the entire world is known by all players at all times. These include Chess, Othello and Go. One characteristic of these games is that the entire game tree can be theoretically traversed with a fast enough computer. While this is never the case in real life (except in simple games like tic-tac-toe), we can create very good results using alpha-beta pruning [15].

Imperfect Information. These games are ones where certain information is withheld. For example in card games one does not know what card will be chosen, or in Scrabble the opponents' letters are kept hidden. Other examples include Spoof and Bridge. A problem facing these games is that a tree based search algorithm has to evaluate the nodes of each possible outcome. This causes a huge branching factor that would be unrealistic to search though. So, in this case a Monte Carlo approach may be well suited [9].

The next deciding factor in catagorising games is its game-tree complexity. Table A.1 shows how the complexity differs greatly between games. This makes a large impact on what algorithms can be used effectively for a game. The algorithm complexity is $O(B^L)$ for a tree search based algorithm, but only $O(N \cdot B \cdot L)$ for Monte-Carlo approaches, where B = average branching factor of the game, L = average game length and N = number of random games [2]. A tree search based algorithm becomes increasingly difficult as the complexity grows, but a Monte Carlo algorithm is less affected by this.

Table A.1: Game Complexity

Game	Checkers	Othello	Chess	Go
B^L	10^{32}	10^{58}	10^{123}	10^{360}

By classifying all games into either perfect or imperfect information and either high complexity or lower complexity, we can compare which types of games a Monte Carlo approach works well at.

Another factor that determines the performance of a monte carlo algorithm is the accuracy of the random simulations [10]. If moves are chosen completely randomly each simulation is less likely to represent an actual game played from this point. What we can instead do is use a heuristic function to decide what move to make each time. For example, in a game like chess or othello we could play a move that maximises our mobility. These intelligent random simulation will give us a better results since the games played will be closer to real games. Hence we will need a smaller number of simulations for the same results. We are interested in how much performance increase we can get from using different heuristics.

Aim

The aim of this project is to find out which games a Monte Carlo algorithm performs best at. Specifically these aims include:

- Develop a gaming platform on which we can build and use agents for comparisons.
- Classify different features of games.
- Evaluate the performance of the Monte Carlo Algorithm against traditional tree-search based algorithms.
- Evaluate the performance of the Monte Carlo Algorithm across different types of games.
- Explore different features of games (e.g. long-term vs short-term strategic moves) to understand the applicability of Monte Carlo approaches to games.

- Explore the effects of using intelligent random simulation with different heuristics.

Method

The task will involve these steps

1. **Background**
   This will involve investigating and classifying games into either perfect or imperfect information and either high or low complexity. Research will also need to be conducted into the current techniques for optimal game strategy.
2. **Game Implementation**
   One or two games need to be developed that are capable of implementing several different AI algorithms.
3. **Algorithm Implementation**
   This will involve implementing two types of algorithms:
   (a) Monte Carlo will be implemented for testing and evaluation across different games. This includes the implementation of heuristics during random simulations.
   (b) MiniMax will be implemented for comparative purposes.
4. **Testing**
   Here experiments will be conducted to compare the performance of the Monte Carlo algorithm against alternative algorithms.
5. **Results**
   The results will then need to be summarised and my dissertation composed.

Software and Hardware Requirements

Only basic requirements, such as access to a lab computer running linux will be needed.

Bibliography

[1] Auer, P., Cesa-Bianchi, N., and Fischer, P. Finite-time analysis of the multiarmed bandit problem. *Machine Learning 47*, 2/3 (2002), 235–256.

[2] Bouzy, B., and Chaslot, G. Monte-carlo go reinforment learning experiments. *IEEE* (2006), 187–194.

[3] Bruegmann, B. Monte carlo go.

[4] Buro, M. The evolution of strong othello programs.

[5] Bushnell, N. Relationships between fun and the computer business. *Commun. ACM 39*, 8 (1996), 31–37.

[6] Canosa, R. Roxanne canosa homepage. http://www.cs.rit.edu/ rlc/.

[7] Dutré, P., Jensen, H. W., Arvo, J., Bala, K., Bekaert, P., Marschner, S., and Pharr, M. State of the art in monte carlo global illumination. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes* (New York, NY, USA, 2004), ACM Press, p. 5.

[8] Fong, G. Adapting cots games for military simulation. In *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry* (New York, NY, USA, 2004), ACM Press, pp. 269–272.

[9] Frank, I., and Basin, D. A theoretical and empirical investigation of search in imperfect information games. *Theoretical Computer Science 252*, 1–2 (2001), 217–256.

[10] Gelly, S., Wang, Y., Munos, R., and Teytaud, O. Modification of uct with patterns in monte-carlo go. *Technical Report 6062* (November 2006).

[11] Gordan, D. Don't forget girls in the effort to close the digital divide. http://www.womensmedia.com/new/girls-tech-Gordon.shtml.

[12] Kocsis, L., and Szepesvari, C. Bandit based monte-carlo planning. *ECML* (2006).

[13] Pleva, G. Game programming and the myth of child's play. *J. Comput. Small Coll. 20*, 2 (2004), 125–136.

[14] Ross, K. W., Tsang, D. H. K., and Wang, J. Monte carlo summation and integration applied to multiclass queuing networks. *J. ACM 41*, 6 (1994), 1110–1135.

[15] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[16] Sklar, E., Parsons, S., and Stone, P. Using robocup in university-level computer science education. *J. Educ. Resour. Comput. 4*, 2 (2004), 4.

[17] Unknown. Othello history. http://home.nc.rr.com/othello/history/.