QUEEN MARY, UNIVERSITY OF LONDON
SCHOOL OF ELECTRONIC ENGINEERING AND COMPUTER SCIENCE

Investigating High Level Monte Carlo Methods

Final Report

Siame Rafiq
Computer Science BSc
s.rafiq@se10.qmul.ac.uk
Student Number: 100647745

SUPERVISOR: DR. SOREN RIIS

22/04/2013

Abstract

Monte Carlo simulations (MC) have been widespread in their application in decision making; especially in competitive games. This study investigates the topics surrounding the effects of simulating Monte Carlo simulations with further Monte Carlo simulations, rather than random simulations. By doing so we add another level to the simulation, giving the name High Level Monte Carlo Method (HMC).

Using the Double Step game as a test case, we investigate the effectiveness of HMC against MC, as well as against itself. This study also serves to gain an understanding of which decisions are suggested, and at which positions in the game they are suggested. This gains an insight into how both the position of the Player on the board is relevant, as well as the position of the opponent on the board.

The data collected from the experiments shows HMC to be an improvement over MC, moving past the pitfalls that have been noted in prior studies (Althofer, 2008)(Ramanujan et al. 2010)(Browne et al. 2012). Although we noted on some suboptimal (odd numbered) squares the performance of the algorithm dropped, we see that HMC can better decipher signals in large games, ensuring a high signal-to-noise ratio, at over 97% in most positions, an increase of up to 18% compared to MC(n). We also note the ruthless approach HMC suggests in its simulations, providing optimal decisions no matter what the distance between the Player and its opponent. This results with over 95% probability of being suggested an optimal move, even with a single iteration of the HMC(n) algorithm; an increase of up to 20% optimal decision selection over MC(n).

We therefore see HMC(n) as an effective improvement over MC(n), with an almost perfect probability of providing optimal decisions in all positions in the Double Step game.

Table of Contents

1. Introduction

In this section I will introduce my motivation behind selecting this topic for my final year project. I will also display my aims for the duration of the project, and how I will outlay my findings in this report.

1.1 Motivation

The Monte Carlo Laziness behaviour as presented in (Althofer 2008) showed an intriguing result, that when a simulation was performed using the Monte Carlo methods, it would become lazy and less focussed on winning if it already had a lead on its opponent. In fact, the algorithm would perform most successfully when the situation was either close or it was somewhat behind.

After gaining an understanding of how these methods work, I wondered what the effect of recursively calling the Monte Carlo decision making algorithm would have on this phenomenon. What interested me more so, was the scope of possibility using these repetitive calls. Not only could the laziness behaviour be prevented, but I was sure that other problems could be optimised as well. An interesting thought was the effect it would have on the speed at which the simulation would make entirely perfect moves, and how many iterations of the simulation would need to be performed before achieving such a state. Also, it would be interesting to see at what stage the High Level Monte Carlo Method becomes affected by this laziness phenomenon, if at all.

Also, by investigating the relationship between iterations and optimal moves, it can also further be investigated the point at which in the game the signal-to-noise ratio becomes high enough that all moves that are made are optimal.

In this report I will investigate how High Level Monte Carlo methods affect efficiency, effectiveness, and how well they work in providing optimal decisions to the computer.

1.2 Aims and Objectives

The aim of this project is to gain a better understanding into the behaviour of High Level Monte Carlo methods. This will be done through performing well known experiments that have been completed using Monte Carlo methods, but in this case using High Level Monte Carlo (HMC) methods. This will provide a baseline to determine the effectiveness of HMC.

The objectives in this project are as follows:

- Research what Monte Carlo methods have been used for, and around the research of Monte Carlo Tree Search.
- Understand the role of Monte Carlo methods used in decision making, and how they can be optimised.
- Build a prototype that can be used to create and simulate the Monte Carlo Method, observing and verifying the results.
- Correctly and efficiently implement High Level Monte Carlo methods within the game, allowing the user to view the impact it has on their game.
- Collect and analyse data from the game and use this to gain a better understanding of the behaviours of High Level Monte Carlo methods.
- Understand whether High Level Monte Carlo methods still suffer from the pitfalls seen in Monte Carlo Methods.

1.3 Report Structure

This report will be structured so that it is clear to read as well as in chronological order with respect to how I approached the problems I faced. Overall, the project will be split into a number of categories, as follows:

Introduction

In this section I will introduce the project that I have undertaken. This will include my understanding before taking on the project, my motivations towards taking this topic and also how this project may solve an actual problem and be a useful piece of research.

Critical Literature Review

The literature review is used to assess what I have found so far from reading sources close to the project I am completing. Whether this is around the topic of Monte Carlo methods or simply on the principals of Java (the programming language I have chosen to complete this project in), this section will look into a deeper view of the surrounding areas of the project and the reasons why it has been so successful.

Program Design

Here I will show the top level design of my program, and how it has been adapted to the task it should perform. This will show both the programming side of the project, as well as the problems faced when attempting to design a program to be encapsulated in such a way that it is compatible with a number of different games.

Results

This section will be mainly tables and figures, outputting the results I have received from the program in different situations.

Analysis

Using the Results section, this section will be used to analyse and investigate the results obtained. Also, this category will be used to display any comparative graphs that could further demonstrate any hypotheses being tested.

Conclusions

From both the Results and Analysis categories, this section will be used to compact all of that information into what can be drawn from this project. Hopefully this will also be used to definitively confirm or contradict any hypotheses put forward. I will also discuss any lessons learnt which may prove useful for any future implementations.

References

Here will be a compilation of all of the references used throughout the final report, structured according to the Harvard referencing system.

2. Critical Literature Review

In this section I will discuss my findings from the literature I have read around the topic of Monte Carlo methods, and how I have chosen the topic of this project from these readings.

2.1 Monte Carlo Method

The Monte Carlo method is an algorithm used to compute results using random sampling. Initially outlined in the 1949 paper titled The Monte Carlo Method by Nicholas Metropolis and Stan Ulam, the paper was an answer to a different approach to the study of differential equations. By using a defined domain of possible inputs, the Monte Carlo Method was to randomly generate inputs within that domain, compute a deterministic result from those inputs, and then to summate all of the results. With the equipment being used at the time of the paper, namely the ENIAC (Electronic Numerical Integrator And Computer), this method proved to be an effective way to solve and study differential equations (Metropolis, 1987). However, the usefulness of the Monte Carlo Method was limited at the time due to the technology available (in 1987) and the resources required; therefore it has only recently come into the forefront of simulation techniques. Having combined itself with effective methods of decision making such as back propagation (Chaslot et al. 2008) and Markov Chains (Brooks 1997), we now see the developments made from the original Monte Carlo Method.

In the most simple of examples, the double step game as described in Althoefer (2008) can be used to display both the strengths and the limitations of the Monte Carlo Method. There have been conjectures provided that can move past these limitations (Baudis, 2011), however for the topic at hand we will focus on the basic Monte Carlo Method and its usage within Monte Carlo Tree Search to provide decision making.

Overall, we can see that the Monte Carlo method is very broad in its applications for simulations, as it is currently implemented in various fields ranging from image processing and graphics (Szirmay-Kalos, 2008), all the way to mapping the evolution of galaxies (MacGillivray, 1982). The simplicity of the initial method stated in The Monte Carlo Method (Ulam, 1949) allows large quantities to be estimated with high accuracy when increasing the amount of randomly generated inputs. This will be further investigated within this project with regards to the way in which the relationship between the number of estimations (iterations) affects the decision which is suggested.

2.2 Monte Carlo Tree Search

The application of Monte Carlo methods that this project will be focussing on is its role in decision making. A particular branch of Monte Carlo methods that has been successful in this area is known as Monte Carlo Tree Search (MCTS), whereby it "combines the precision of tree search with the generality of random sampling" (Browne et al 2012). Proposed by Remi Coulom in 2006 (Coulom, 2006), MCTS provided an novel way to simulate all the decisions from the current situation to the end of the scenario, selecting each node on its probability that it was better than the current best decision.

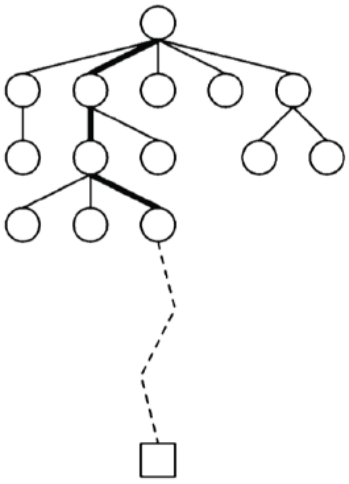

Figure 1. Basic MCTS Process (Baier 2010)

In Figure 1, we can see the basic outline of the MCTS process. All of the decisions from the current node are expanded, the best route is taken, and the method is repeated until the goal state is reached.

Having been so successful within the realm of correctly simulating and performing at the game Go (Brugmann, 1993), MCTS has been effective at decision making in terms of both deterministic and non-deterministic situations.

By randomly simulating all decisions from a certain situation, MCTS selects the decision with the highest level of success, using this to progress the aforementioned situation. MCTS will be the basic form of decision making implemented within this report, in order to simulate and observe any effects that are carried through on a higher level.

As with any simulation, certain limitations have been brought to light within MCTS. There is no definitive guarantee that an extension in the amount of iterations used to simulate the decision will result in the best decision being made (Kocsis et al, 2006). This means that the simulation is not guaranteed in terms of game-theory, as it cannot produce the optimal result with exact accuracy. This provides some reasoning for the interesting results found within the laziness phenomenon (Althofer, 2008), which leads us to further investigate at which stage the signal-to-noise ratio becomes high enough that there is a game-theoretic guarantee within the simulation. In other words, how far along the simulation does it provide optimal moves as its suggestion, and how does the amount of iterations used to simulate that decision affect this.

Having no heuristic guide, MCTS provides the best decision at any stage on a purely probabilistic sense, showing its range and adaptability to many different situations. Having shown its strengths from Scrabble (Sheppard, 2002) to Go (Brugmann, 1993), MCTS will be

used in this project for the double step game, in which it will be observed under a number of conditions.

In the version of MCTS as described in (Chaslot et al., 2008) there are 4 stages to the process. In Figure 2, we can see the entirety of the MCTS algorithm. In the Selection state, the tree is traversed recursively until the most available expandable node that has not been expanded is reached. This means that the node is not the end of the scenario, as well as has not been expanded further. During Expansion, the nodes are then expanded from that expandable state. In Simulation, the entirety of the scenario is simulated from that expanded node until the end of the scenario is reached, and then in Backpropagation, the result is effectively "fed" up the tree and the probabilities for each of the nodes is updated. (Browne et al., 2012)

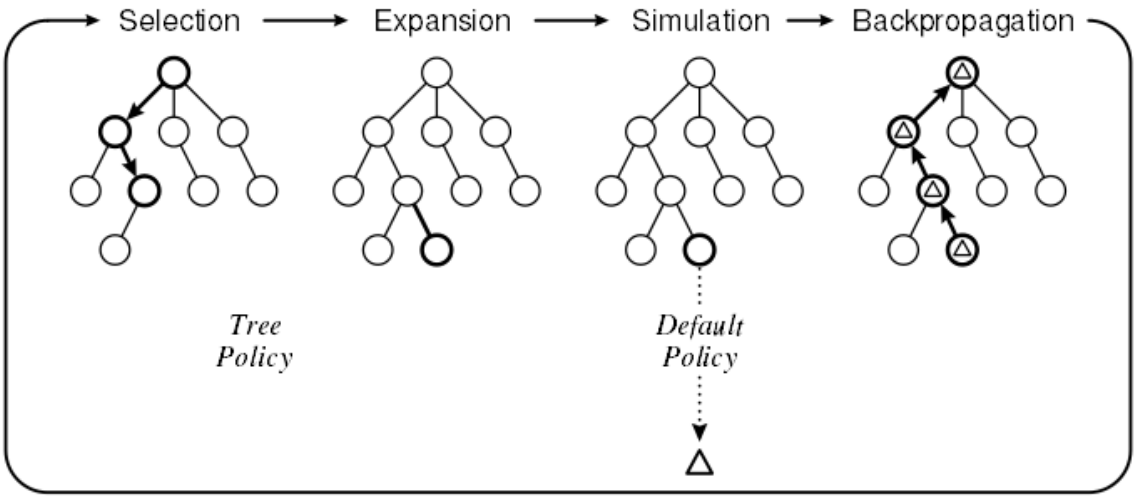

Figure 2. An iteration of how MCTS works (Browne et al, 2012)

In terms of this project, the selection step will be getting the simulation up to the stage that each player is at. Expansion is then adding each possibility from that stage (i.e. move one step forward or two steps forward). Simulation will be to complete the game from there using random sampling, and back propagation will add to a counter whether or not the player won the game, effectively updating the probabilities of the decisions that the player can make. Using this outcome, the player will then choose to move one, or two steps forward.

Browne (Browne et al, 2012) describes the true depth in which the MCTS algorithms have been put to work, listing both deterministic and non-deterministic games and situations.

2.2.1 Deterministic

Deterministic situations occur when for any given input, the output is the same every time. This is prevalent in games in which all inputs can be seen by all parties such as Chess or Tic Tac Toe. These types of games are excellent benchmarks for the MCTS algorithm as they can be run and observed under different and explicitly set conditions.

This includes the double step game, which wil be used in this project. Having shown the laziness phenomenon on its simplest scale (Althofer, 2008), this game is a usable test case where we can experiment with the Monte Carlo and High Level Monte Carlo Methods. By also being applicable to developments to MCTS as described in (Kocsis et al, 2006)(Ramanujan et al, 2010), the double step game has been chosen to test the High Level Monte Carlo methods on, in order to observe its behaviours.

2.2.2 Non Deterministic Situations

Non deterministic situations are when there is a random element to the game, and the same situation may bring about two different results. Monte Carlo methods are particularly effective with non deterministic situations as they are able to determinise the situation, by "fixing the outcomes of all chance events and making states fully observable. For example, a card game can be played with all cards face up, and a game with dice can be played with a predetermined sequence of dice rolls known to all players." (Brown et al 2012). This further shows how by using random sampling with a consistent seed the MCTS is able to provide sound decisions even in a situation with a random element to it.

Another way in which non deterministic situations can be mapped are through using Multiple MTCS (MMTCS) as shown in (Auger, 2011). Using this multiple trees are stored and updated with each move. When making a decision for a player, that specific player's tree is searched at each stage, allowing the search to "more accurately models the differences in information available to each player than searching a single tree." (Browne et al 2012). This allows each player to have their own tree reflecting the information that is available to them at that time.

As the double step game is deterministic in nature, there is no need to implement MMTCS (Auger, 2011), as all players can see the state of all other players at any time on the board.

2.3 Monte Carlo Lazy Behaviour

After understanding the effectiveness of the MCTS, it was now interesting to see how it performs in different situations. Althofer (2008) describes it as being "lazy", and that "Our experiments indicate that MC is at its best when the board situation is balanced or even slightly worse for the player to move.", showing the MCTS as more likely to select a non-

optimal move when it feels it is in a position too far from its opponent. This is down to the ineffectiveness of that decision on the end result of the game, as the simulations return that there is no real difference between one move or another (Althofer, 2008). Again, this is shown in the table (Table 1 (Althofer 2008)) below. This is another hypothesis that I will investigate within this project.

		Board Size							
Monte Carlo Simulations		6 vs 3	6 vs 4	6 vs 5	6 vs 6	6 vs 7	6 vs 8	6 vs 9	6 vs 10
	1	0361	2204	4805	6933	9340	9649	9967	9988
	2	0193	1987	5799	6970	9654	9752	9991	9996
	4	0047	1468	7039	7450	**9905**	**9868**	9999	9999
	8	0002	0836	**8573**	**8228**	**9989**	**9969**	**10000**	**9999**
	16	0000	0286	**9557**	**9264**	**10000**	**9992**	10000	10000
	32	0000	0040	**9936**	**9875**	10000	10000	10000	10000
	64	0000	0000	**10000**	**9991**	10000	10000	10000	10000

Table 1 – Number of wins for computer (Althofer 2008)

In Table 1, the highlighted cases are those in which the computer most often wins when the game is close, due to being either behind or on par with the other player.

As the Monte Carlo Method relies purely on random sampling, the reasoning behind the laziness phenomena is in its effect on the end game. When the game is not tight, all moves become probabilistically close at either end of the spectrum. In the case of the double step game, if the player has a lead on its opponents, all moves have a probabilitic value close to 1, whereas if the player is behind, all probabilistic values are close to 0 (Althofer, 2008). This means that no matter what move the simulation makes, they will almost always win or lose, respectively.

In terms of this project, this raises the interesting question as to how quickly this “laziness” will arise when performing the High Level Monte Carlo method. Also, how will the opposing player react to a computer with optimal movements all of the time; how quickly will their moves become lazy? I will further analyse the point at which suboptimal moves begin to be made, in order to gain a further understanding of when laziness sets in, if at all.

2.4 Signal-to-noise ratio

At the beginning of any decision tree, in the cases of non-perfect play, decisions made have much less of an impact on the rest of the scenario. Althofer, (Althofer, 2008) shows that making an incorrect move against an opponent is detrimental to the probability of winning, but relative to the positon at which the sub-optimal move was made.

In terms of signal to noise, this results in the beginning of the scenario having many different possibilities, however only a slight affect on the end result of the game when compared to decisions made at the end of the scenario (Browne et al, 2012). This however only applies to games with few or little trap states (i.e. states where you can lose after a few moves, for example Chess), as shown in (Ramanujan et al, 2010). In games with higher amounts of trap states, it can be seen that decisions made at the beginning of the game have an adverse effect of the result, as the possibility of losing is not far away.

In terms of this project, the double step game will be lengthened such that there are no prevalent trap states within the decision tree (i.e. a suboptimal initial move almost guarantees failure). This will also provide a platform where the relationship between the amount of iterations of the High Level Monte Carlo method can be analysed with respect to the distance from the end of the board before all moves that are made are optimal. This will show when the signal-to-noise ratio is high enough whereby each move is considered game-critical and requires to be optimal in order to win the game.

2.5 Enhancements to MCTS

The MCTS algorithm can be further enhanced by using added principles depending on the target program. This could be in the way of taking into account other domain knowledge, from either previous winning games or a store of best strategies. The algorithm could even be adapted to learn from its own victories and losses, providing weighting to each decision made, thereby adding to the effectiveness of the overall recommendation. In this case, Drake (2007) described such games as “heavy playouts”. Alongside this, debate has been struck about whether it is more efficient to optimise the tree searching policy or the Monte Carlo simulation policy. Heavy playouts were tested but in the end it was found to be more efficient to improve the Monte Carlo simulation policy in order to gain a better recommendation and decision (Drake 2007).

Upper Confidence Bound for Trees (UCT) (Kocsis et al, 2006) has been proven to be able to overcome the non guarantee game theoretic decision suggested by the MCTS algorithm. This shows that given enough time, when applying UCT on top of MCTS, it is possible for the algorithm to converge to the minimax tree values (i.e. optimal decisions are made at any decision making situation). However, minimax can only be seen as advantageous over MCTS when heuristic measures are in place, otherwise MCTS can be seen as the more applicable tree search.

Furthermore, UCT can be ineffective depending on the domain. As shown in (Ramanujan et al, 2010), when there are many trap states in the scenario, the minimax outperforms UCT. In the context of this project, the length of the double step game will need to be varied in order

to assure that there is no real prevalence of trap states. A shorter double step game can be lost in only a few moves, whereas with a longer board it will be impossible to win in few moves as $n/2$ board moves are required where n is board length.

When considering these developments in terms of the topic of this report, we will hope to see increases in the optimality of decision made by the High Level Monte Carlo Methods.

2.6 Applications in Decision Making

With the initial Monte Carlo Method being applied to the Computer game Go (Brugmann, 1993), the adaptation to the Monte Carlo way of simulation within many different branches of decision making has been widespread. Chaslot and Bakkes (Chaslot et al, 2008) were pioneers in this area, scribing MCTS as a framework to other decision making problems within Artificial Intelligence (AI).

Gelly (Gelly et al, 2006) further contributed to the progression that the Monte Carlo method had within the game Go, getting it to a level where it achieved master status on a 9x9 Go board (Lee et al, 2009).

However, the pitfalls of MCTS have shown through in a game like Chess (Ramanujan et al., 2010). When compared to the Minimax tree search algorithm (Russell & Norvig, 2010), Chess is much more in favour of the careful nature of Minimax, where it is aware of the initial search traps (Ramanujan et al., 2010). This will be taken into account when the MCTS algorithm is implemented for the double step game, and investigated further by looking into at which point a suboptimal move causes almost certain defeat.

All these progressions show that the field of decision making in games within the Monte Carlo Method is moving forward at a great speed. The only constraint of the Monte Carlo method when it was first proposed, was the capability of the computers that were available. (Metropolis, 1987).

2.7 Dynamic Komi

A recent development in the progression to combat the laziness phenomenon (Althofer, 2008) has been suggested by constantly rebalancing the end goal of the program. Known as Dynamic Komi, it has been shown to provide an effective way to combat laziness (Baudis, 2011). By constantly re-evaluating the end goal in a way that keeps the situation competitive no matter what the lead or deficit encountered by the player, this attempts to always keep the game in a state that is close where the Monte Carlo Method has proved so successful (Browne et al, 2012)(Althofer 2008).

For example, if moving first in a two player game gives the player an obvious advantage, then that player must not only win the game, but win by the margin of their first move also. In doing so, this always keeps the first player on edge, as any non-optimal move (considering that the other player also makes optimal moves) means that they would lose the game.

In the case of this project it will be interesting to see first if there are any remains of the laziness phenomenon when subjecting the decision to High Level Monte Carlo methods. If so, it should follow along the lines from (Baudis, 2011) that “Dynamic Komi” should prove an effective way to ensure the game remains in a tight situation throughout.

2.8 High Level Monte Carlo Methods

Now that we have seen the effectiveness of Monte Carlo methods on games, this project will focus on how to enhance these results. Intuitively, these algorithms provide random sampling at the first level, meaning a decision is put forward, random sampling is done on the remaining situations to see which is probabilistically the most likely for success and then that course of action is taken (Coulom, 2006).

Within this project I will be attempting to see the effects of making the randomly sampled simulations also subject to Monte Carlo methods, and noting the changes this makes. I hope to understand with reference to high level Monte Carlo methods the following;

- Whether a different number of simulations are required to obtain the same level of success within the simulations preferred decision making.
- At which stage does the simulation provide optimal decisions at each node in the tree, (i.e. how far from the end before all decisions that are made are optimal).
- Overall what the effect is of High Level Monte Carlo Methods when compared to basic Monte Carlo methods based on the decisions made by the simulation.

3. Program Design

In this section I will describe how the system was designed and built, henceforth showing the process that was implemented to ensure that the system was functioning correctly. This will also demonstrate the way in which the system will collect and display results.

3.1 System Objectives

After performing my background research I have decided on the following objectives for the system to be able to carry out in this experiment:

- To be able to change the board size for either player
- To be able to have either player using one of the three types of simulation (Random, MCM (Monte Carlo Method), HLMCM (High Level Monte Carlo Method))
- To provide the number of wins in the actual Double Step Game from any number of iterations
- To provide statistics on the probabilities of which decision is suggested via the Monte Carlo and High Level Monte Carlo methods and at which position these were suggested

3.2 System Overview

The required simulation is simple when looked at from a high level design perspective. All it requires is a class that is used to hold each Player, and a class that can be used to represent the Double Step Game. These classes are then run using a driver class, which manages and holds all of the Game and Player classes. A helper class, the Decision class, is used to store statistics about which decision was suggested and which position it was suggested at. A general overview of the program can be seen in the UML diagram below:

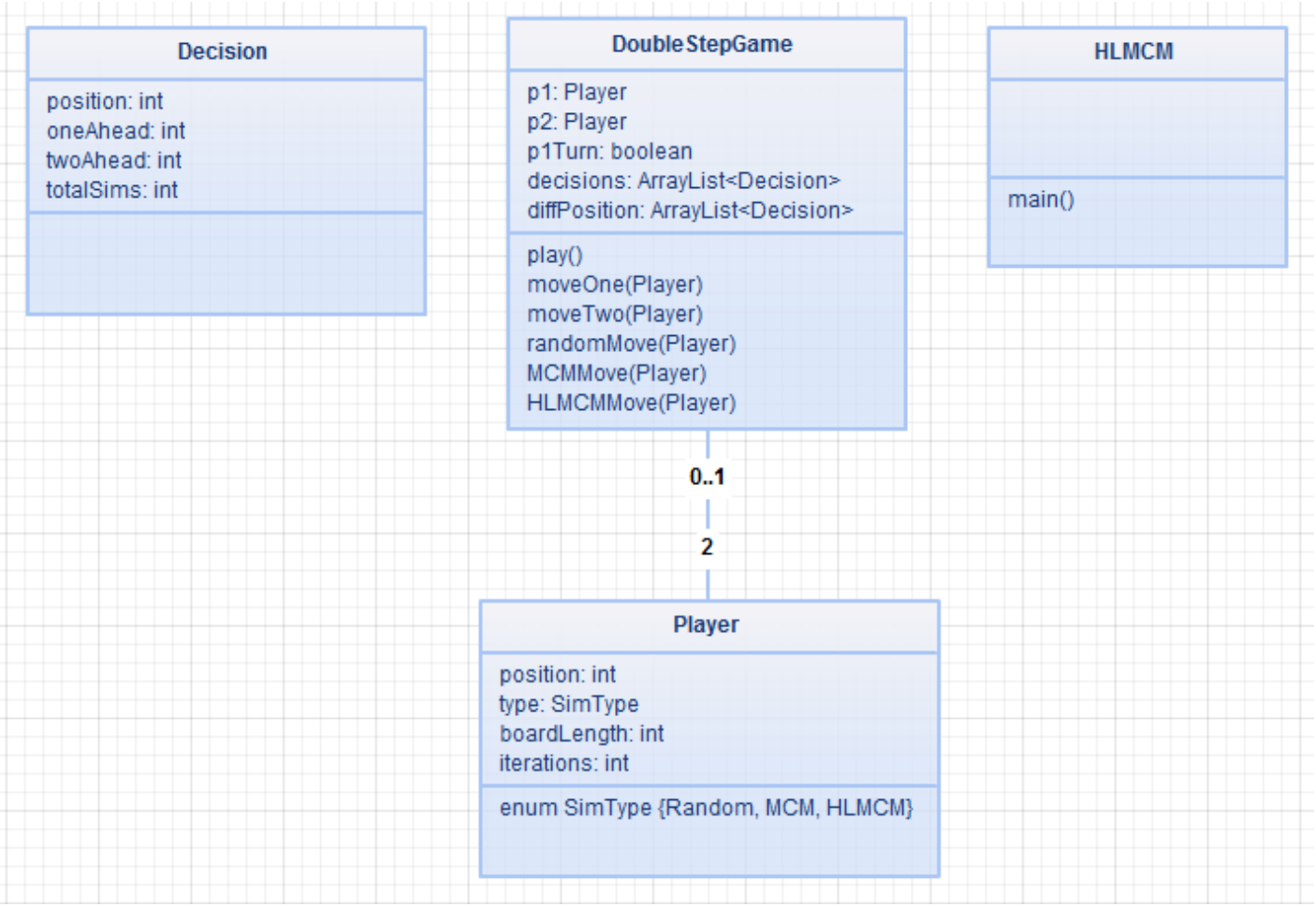

Figure 3 - UML Diagram of the System Used

3.2.1 Player

The Player class stores the position of itself, as well as the length of the board it must travel to win. As well as this, the player also stores the type of simulation it uses to make its decisions. This is an enumerable; Random, MCM or HLMCM. By creating the Player class as its own class and not a nested class, this keeps a sense of modularity and extendibility. This allows the class to be easily built on if needed for other more advanced games where more information may need to be stored about the Player.

In addition to these storages within the Player class, there is also one method that is required (apart from the usual getters), and that is to modify the position. In the case of the double step game it is used to increment the Player's position by either one or two. The exact game logic is kept separate from the Player class, in order to further the case of modularity.

3.2.2 Game

The Game class requires more information to be stored than the Player class. At the most primitive level, it requires 2 players and some sort of method to operate the game, in which the game logic will be stored.

In my implementation, I have written the Game to have one method to operate the games logic. The play() method is used to direct the flow of play, and perform each simulation with respect to the Player's simulation type. It will then provide, as an output, the Player who won that game. This method will call either randomMove(), MCMMove(), or HLMCMMove(), depending on the Player's simulation type. It will also ensure that one Player does not have two turns in a row, and alternate game flow in that sense.

Each of the move methods does not access the Player's position by themselves, but through other helper methods, which in this case are moveOne() and moveTwo(). These methods provide a pre-screening to the move being made, ensuring it is valid before actually changing anything on the board (e.g. if a Player is on square 9 of a 10 square board and then rolls a 2, this move is illegal as they would move off of the end of the board. Instead, the turn is passed without any move being made).

The Monte Carlo methods have been programmed such that they should work no matter what the game, subject to the play() method being coded correctly.

MCMMove(n), is used to provide decisions using Monte Carlo methods. HLMCMMove(n) is used to provide decisions using High Level Monte Carlo methods.

The HLMCMMove(n) method is built so that it calls the MCMMove(n) method, which in turn provides one extra level of Monte Carlo processing. This adequately provides the functionality described earlier in this report, whilst still keeping the program as simple and as efficient as possible.

The reason they have been linked this way is to ensure large computations can be completed. Both MCMMove(n) and HLMCMMove(n) work using iteration, rather than recursion. This choice was made because of the large numbers that I will be testing with. When using recursion, it is not long in using HLMCMMove(n) before the environment throws a stack overflow error, or a recursive depth error. This was fixed by switching to iteration. Although this has required more coding and a slightly slower implementation for smaller values of n, this has enabled me to be able to test on large values of n, which will in turn increase the ability to locate correlations in the data.

3.2.3 Decision

The Decision class is used to store the decisions suggested by the MCM and HLMCM. This is in order to gain an understanding of the decision that is suggested with relation to the position of the Player on the board, as well as the position of the opposition on the board. Using this class, we will be able to explicitly analyse if there is any laziness occurring when using these methods.

This will be done by analysing the amount of times the optimal move is suggested out of the total amount of simulations that occurred in that position. Also, the amount of times that the optimal move is suggested in relation to the opposition's position will also be recorded.

Both these statistics will provide a different view on the laziness phenomenon from the amount of wins that are recorded at the end of each game. By gaining an insight into what is suggested at every stage of the MCM or HLMCM, we will be able to see how different factors affect the decision suggested. This includes the amount of iterations for the simulation, the distance from the end of the board and the distance between the two players.

3.3 Double-step game

Using the 2 step game as described in (Althofer, 2008) is an efficient way to judge the effectiveness of High Level Monte Carlo methods. As there have already been studies on the reliability and optimisation of searching through decision with Monte Carlo methods (Drake, 2007), (Brugmann, 1993), the Double Step game presents itself as a simple and efficient way to show any occurring correlations.

The game is as follows: There is a board of length n, and players 1 and 2 start at square 0. Each player can then move 1 or 2 forward towards the final square, n. The decision to move 1 square or 2 squares forward depends on the suggestion from the simulation. The first player to reach the final square wins.

The Double Step Game will work with the Monte Carlo methods as follows: At each decision point, 2 games will be instantiated; one, with the player having moved 1 step forward, and another, with the player having moved 2 steps forward. The games will then be simulated randomly until there is a victor. A number, containing the ID of the victor will be returned to the user (1 for Player 1, 2 for Player 2). The user will then perform this simulation a certain number of times (iterations), and the beginning move that resulted in the most wins, is the move the Player will make in the actual game.

3.4 Game Mechanics

Player 1 is on square 2. Player 2 is on square 3. It is Player 1's turn. Player 1 checks its next best move using Monte Carlo methods, and wants to complete 100 iterations. After the Monte Carlo method is completed, The Monte Carlo method has the values 35, and 60. 35 corresponds to the amount of games won when moving 1 step forward, 60 corresponds to the amount of games won when moving 2 steps forward (both out of 100).

As the Monte Carlo method shows, moving 2 steps forward has a higher success rate than moving 1 step. As a result, the Player performs this move in the actual game. This means

that Player 1 is now on square 4, and Player 2 is on square 3. It is Player 2's turn, and the mechanics of the game are repeated.

With High Level Monte Carlo Methods, this is applied even to the Monte Carlo Method's game. For example, each HLMCM decision simulates its decisions using the Monte Carlo Method and then this uses random simulation. This can be accurately shown using the diagrams below;

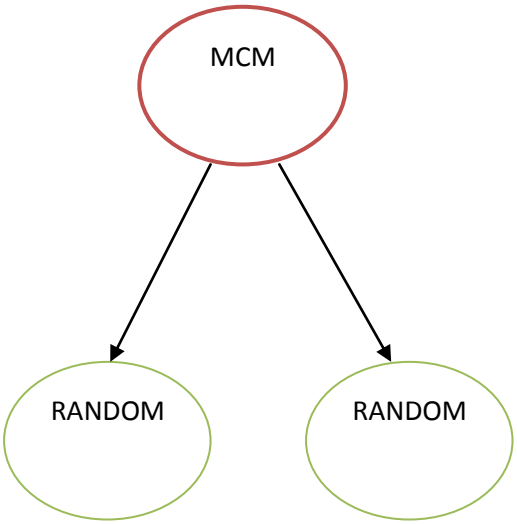

Figure 4 - Monte Carlo Method Simulation

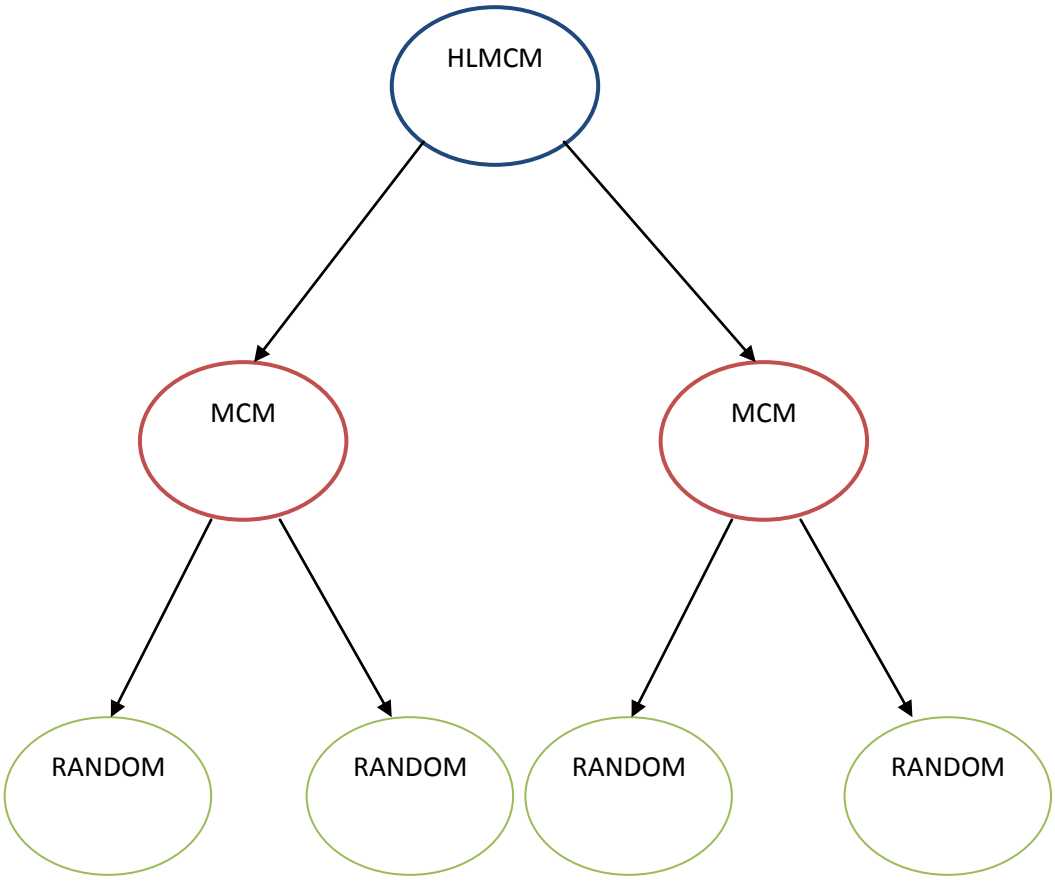

Figure 5 - High Level Monte Carlo Method Simulation

Each decision within the MCM is only simulated using random simulation. HLMCM simulates games by creating 2 MCM games, which then further go on to simulate games randomly. This provides the next level of simulation, giving the name High Level Monte Carlo Method. By doing so, we hope to see increased optimality in suggestions, as well as developments to the Laziness phenomenon (Althofer, 2008).

As stated before, the methods have been coded so that they effectively can be chained together. HLMCMMove() calls MCMMove() which then calls randomMove(). The level of simulation simply depends on where the Player begins the chain at. This has been done in order to keep the game extendable, as well as keep the option open for further investigation of even Higher Levels of Monte Carlo Methods.

4. Results

In this section I will state the results I have gathered from my program when testing it in a variety of situations. For the results obtained, the game used was the Double Step game, as described in Althofer (Althofer, 2008). The board size has been set to 10 as a default for all the below results unless stated otherwise. All results are calculated from 10,000 simulated games.

4.1 Basic Monte Carlo Method

The results below are using only a basic Monte Carlo (MC) method, to establish a baseline against which the High level Monte Carlo method will be competing. This will be done with one player using MC, and the other using either a random system, or also using MC.

4.1.1 Versus Random Opposition

MC(n)	Number of Wins for MC player
1	9765
2	9768
4	9770
8	9763
16	9792
32	9886
64	10000

Table 2 – Number of Wins for MC Player against Random Opposition

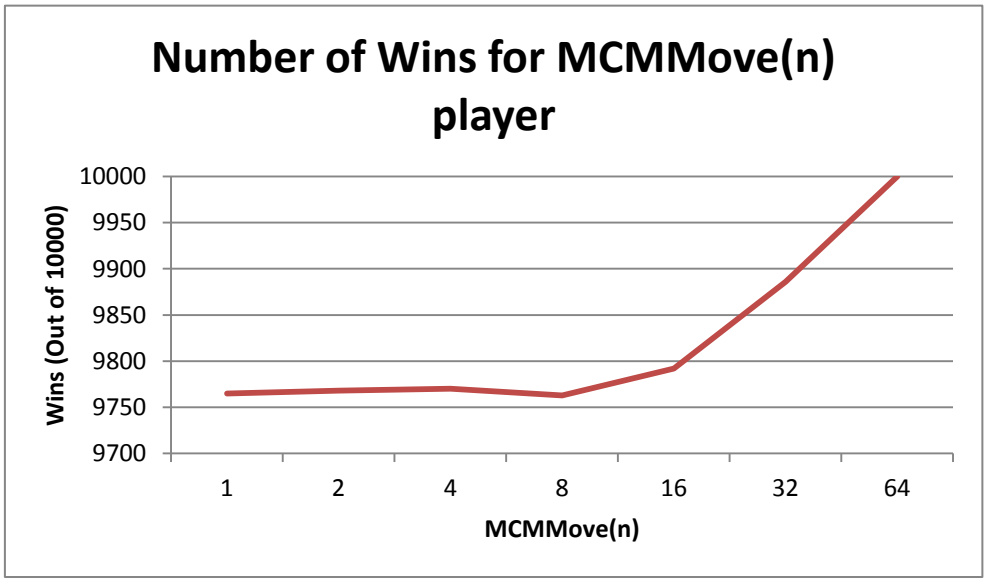

Figure 6 – Number of Wins for MC Player versus Random Opposition, plotted against n (iterations)

4.1.2 Versus Monte Carlo Opposition

As illustrated in Table 3, along the top we have the amount of simulations for Player 1, and across the side we have the amount of simulations for Player 2.

Player 2 \ Player 1	1	2	4	8	16	32	64
1	6870	6881	7278	7215	7905	8780	9563
2	7261	7203	7607	7521	8117	8822	9594
4	7167	7172	7597	7529	8168	8861	9540
8	6803	6876	7358	7189	7861	8713	9506
16	6278	6338	6847	6638	7491	8452	9401
32	5566	5560	6059	6147	6896	8080	9234
64	4713	4885	5437	5314	6346	7722	9122

Table 3 – Number of wins for Player 1 (MC Player) versus Player 2 (MC Player)

4.2 High level Monte Carlo Methods

In this section we will look at how the High Level Monte Carlo Methods (HMC(n)) fare against the other types of simulation; random simulation, Monte Carlo (MC(n)) simulations, and HMC(n) simulation.

4.2.1 Versus Random Opposition

HMC(n)	Number of Wins for MC player
1	9969
2	9981
4	10000
8	10000
16	10000
32	10000
64	10000

Table 4 – Number of wins for HMC Player versus Random Opposition

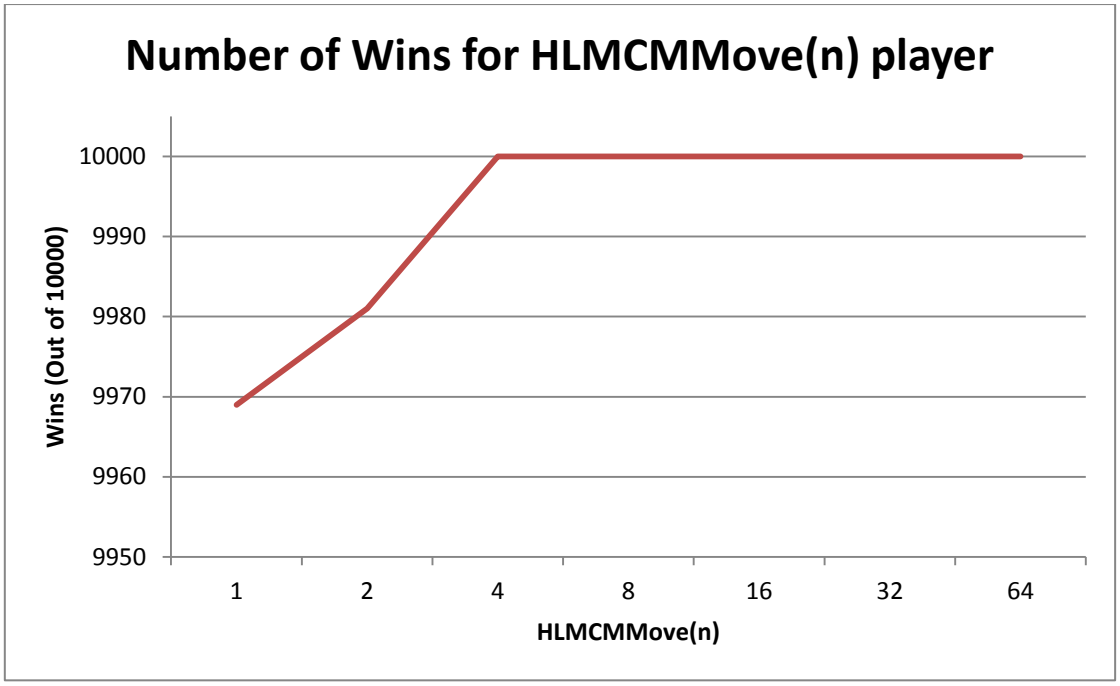

Figure 7 – Number of Wins for HMC Player versus Random Opposition, plotted against n (iterations)

4.2.2 Versus Monte Carlo Opposition

Player 1

Player 2	HMC(n)						
MC(n)	**1**	**2**	**4**	**8**	**16**	**32**	**64**
1	9192	9249	9328	9580	9718	9770	10000
2	9224	9547	9504	9639	9642	9791	10000
4	9267	9366	9425	9585	9749	9795	10000
8	9087	9318	9181	9515	9650	9777	10000
16	8997	9130	9097	9398	9536	9746	10000
32	8536	8952	8864	9298	9499	9685	10000
64	8046	8733	8603	9110	9347	9579	10000

Table 5 – Number of wins for Player 1 (HMC Player) against Player 2 (MC Player)

4.2.3 Versus High Level Monte Carlo Opposition

Player 2 \ Player 1	1	2	4	8	16	32	64
HMC(n)							
1	8876	8926	9045	9364	9687	9706	10000
2	8694	8779	8973	9307	9675	9696	10000
4	8539	8755	8894	9236	9633	9645	10000
8	8400	8694	8747	9166	9595	9611	10000
16	8392	8662	8672	9093	9530	9599	10000
32	8156	8582	8564	9051	9439	9502	10000
64	7902	8555	8438	8991	9358	9487	10000

Table 6 – Number of wins for Player 1 (HMC Player) versus Player 2 (HMC Player)

4.3 Laziness

In this section we will analyse the results when pitting two players against each other, both using High Level Monte Carlo Methods. We will recreate the experiment performed by Althofer (Althofer, 2008), this time using High Level Monte Carlo Methods. In conjunction with this, we will also analyse the decisions made using MCM, and HLMCM, seeing if there is any difference between the moves suggested.

4.3.1 Changing Board Length

HMC(n)	6v3	6v4	6v5	6v6	6v7	6v8	6v9	6v10
1	0	0	9733	9874	10000	10000	10000	10000
2	0	0	9670	9733	10000	10000	10000	10000
4	0	0	9764	9655	10000	10000	10000	10000
8	0	0	9955	9699	10000	10000	10000	10000
16	0	0	9998	9903	10000	10000	10000	10000
32	0	0	10000	9990	10000	10000	10000	10000
64	0	0	10000	10000	10000	10000	10000	10000

Table 7 - Number of wins for Player 1 (HMC Player) versus Player 2 (HMC Player) on a variety of board lengths

4.3.2 Decisions Suggested with regards to Position

In this section, we will use a board length of **50** instead of 10, in order to have a larger window in which there should be a low signal-to-noise ratio. Also, this will allow us to better see the effects of the decision suggested based on position, as there will be a larger number of positions that can be tested (50), instead of the usual 10.

These experiments will be performed using MC(n) and HMC(n), in order to see the difference between the two ways of simulations' decision making.

MC(n)

Position on Board (Out of 50)	1	2	4	8	16	32	64
0	0.7173	0.7296	0.7443	0.7537	0.7928	0.7833	0.8447
1	0.7428	0.7259	0.7438	0.7360	0.7861	0.7849	0.8390
2	0.7196	0.7275	0.7440	0.7542	0.7993	0.7878	0.8501
3	0.7346	0.7257	0.7481	0.7503	0.8067	0.7842	0.8576
4	0.7257	0.7204	0.7399	0.7440	0.8088	0.7961	0.8524
5	0.7277	0.7282	0.7460	0.7535	0.7929	0.7914	0.8574
6	0.7285	0.7333	0.7567	0.7571	0.8077	0.7997	0.8572
7	0.7286	0.7330	0.7536	0.7591	0.7985	0.8028	0.8696
8	0.7262	0.7311	0.7410	0.7584	0.8054	0.7946	0.8678
9	0.7189	0.7344	0.7559	0.7466	0.8090	0.7883	0.8626
10	0.7347	0.7390	0.7516	0.7628	0.8048	0.7931	0.8689
11	0.7279	0.7396	0.7519	0.7617	0.8067	0.8058	0.8702
12	0.7229	0.7401	0.7477	0.7589	0.8101	0.7996	0.8641
13	0.7388	0.7350	0.7627	0.7558	0.8069	0.8002	0.8676
14	0.7428	0.7353	0.7584	0.7689	0.8115	0.8145	0.8716
15	0.7345	0.7412	0.7656	0.7661	0.8023	0.8084	0.8751
16	0.7495	0.7436	0.7720	0.7818	0.8246	0.8007	0.8659
17	0.7548	0.7433	0.7616	0.7780	0.8116	0.8141	0.8735
18	0.7426	0.7470	0.7737	0.7756	0.8211	0.8125	0.8819
19	0.7472	0.7583	0.7725	0.7723	0.8179	0.8209	0.8751
20	0.7443	0.7454	0.7785	0.7754	0.8255	0.8190	0.8825
21	0.7483	0.7561	0.7715	0.7822	0.8234	0.8222	0.8935
22	0.7453	0.7568	0.7688	0.7898	0.8317	0.8246	0.8852
23	0.7554	0.7563	0.7809	0.7874	0.8240	0.8253	0.8866
24	0.7484	0.7641	0.7845	0.7898	0.8263	0.8265	0.8892
25	0.7657	0.7665	0.7757	0.7921	0.8270	0.8246	0.8915
26	0.7469	0.7668	0.7773	0.7960	0.8353	0.8363	0.8970
27	0.7500	0.7649	0.7865	0.7851	0.8490	0.8288	0.8983
28	0.7668	0.7770	0.7801	0.7945	0.8399	0.8307	0.9043
29	0.7706	0.7589	0.7854	0.7947	0.8381	0.8342	0.9056
30	0.7636	0.7632	0.7866	0.8035	0.8451	0.8329	0.9012
31	0.7724	0.7843	0.7888	0.8039	0.8399	0.8408	0.9094
32	0.7797	0.7730	0.7890	0.8014	0.8500	0.8436	0.9100
33	0.7713	0.7811	0.7933	0.8061	0.8475	0.8420	0.9213
34	0.7773	0.7789	0.8044	0.8113	0.8500	0.8377	0.9095
35	0.7817	0.7892	0.8100	0.8085	0.8534	0.8545	0.9097
36	0.7902	0.7975	0.8015	0.8165	0.8582	0.8499	0.9197
37	0.7837	0.7859	0.8063	0.8323	0.8628	0.8526	0.9182
38	0.7976	0.7976	0.8067	0.8191	0.8622	0.8507	0.9152
39	0.7956	0.7969	0.8075	0.8246	0.8612	0.8552	0.9194
40	0.8028	0.8021	0.8073	0.8375	0.8720	0.8627	0.9245
41	0.7964	0.7978	0.8241	0.8358	0.8702	0.8556	0.9227
42	0.8046	0.8063	0.8085	0.8413	0.8782	0.8759	0.9305
43	0.7842	0.8100	0.8151	0.8386	0.8744	0.8615	0.9236
44	0.8182	0.8173	0.8435	0.8486	0.8837	0.8858	0.9387
45	0.7745	0.7899	0.7890	0.8193	0.8719	0.8178	0.8881
46	0.8729	0.8691	0.9056	0.8738	0.9010	0.9315	0.9712
47	0.8602	0.8692	0.8888	0.8869	0.9138	0.8858	0.9802
48	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
49	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Table 8 - Probability of selecting the optimal move at any position on the board with the move suggested by MC(n) where n is iterations

HMC(n)

Position on Board (Out of 50)	1	2	4	8	16	32	64
0	0.9740	0.9810	0.9900	0.9910	0.9994	1.0000	1.0000
1	0.8846	0.9473	1.000	1.0000	1.0000	1.0000	1.0000
2	0.9733	0.9867	0.9191	0.9949	0.9989	0.9800	1.0000
3	1.0000	0.9677	0.8888	0.8571	1.0000	1.0000	1.0000
4	0.9726	0.9824	0.9565	0.9949	0.9984	0.9897	1.0000
5	0.9333	0.9787	1.0000	1.0000	1.0000	1.0000	1.0000
6	0.9677	0.9884	0.9886	1.0000	0.9988	0.9896	1.0000
7	0.9300	0.9649	1.0000	0.9411	1.0000	1.0000	1.0000
8	0.9735	0.9830	0.9885	0.9928	0.9987	0.9687	0.9900
9	0.8803	0.9436	0.8571	1.0000	1.0000	0.8571	1.0000
10	0.9732	0.9785	0.9886	0.9918	0.9980	0.9893	1.0000
11	0.9291	0.9080	0.9230	1.0000	0.9863	1.0000	1.0000
12	0.9614	0.9739	1.0000	0.9896	0.9993	0.9892	1.0000
13	0.9342	0.9320	1.0000	1.0000	1.0000	0.8750	1.0000
14	0.9848	0.9745	0.9318	0.9885	0.9975	0.9892	0.9898
15	0.9096	0.9663	0.8888	1.0000	1.0000	1.0000	1.0000
16	0.9790	0.9706	0.9642	0.9915	0.9977	0.9891	1.0000
17	0.9245	0.9290	1.0000	0.9833	1.0000	1.0000	1.0000
18	0.9683	0.9792	1.0000	0.9957	0.9972	0.9560	1.0000
19	0.9367	0.9597	0.9473	0.9523	0.9933	1.0000	1.0000
20	0.9701	0.9824	0.9756	0.9904	0.9975	0.9885	1.0000
21	0.9255	0.9430	0.9500	0.9710	0.9827	0.9285	1.0000
22	0.9769	0.9847	0.9876	0.9914	0.9966	0.9770	0.9897
23	0.9170	0.9629	0.8500	0.9866	0.9950	0.9333	1.0000
24	0.9744	0.9774	0.9759	0.9892	0.9975	0.9883	1.0000
25	0.9040	0.9200	0.9473	0.9404	0.9867	0.9333	1.0000
26	0.9671	0.9797	0.9390	0.9902	0.9953	0.9883	0.9793
27	0.9417	0.9325	0.9130	0.9318	0.9739	0.9333	0.8000
28	0.9851	0.9748	1.0000	0.9901	0.9952	0.9883	0.9791
29	0.9029	0.9251	0.9047	0.9340	0.9837	0.8666	1.0000
30	0.9828	0.9733	0.9629	0.9923	0.9947	0.9425	1.0000
31	0.9450	0.9641	1.0000	0.9347	0.9887	1.0000	1.0000
32	0.9815	0.9790	0.9743	0.9879	0.9943	1.0000	0.9893
33	0.9117	0.9219	1.0000	0.9690	0.9753	0.9444	1.0000
34	0.9864	0.9728	1.0000	0.9922	0.9921	1.0000	0.9784
35	0.9289	0.9573	0.8750	0.8910	0.9765	0.9411	0.8888
36	0.9840	0.9912	0.9620	0.9879	0.9925	0.9761	0.9782
37	0.9387	0.9138	0.8750	0.9306	0.9641	0.8888	0.8000
38	0.9840	0.9851	0.9873	0.9878	0.9936	1.0000	1.0000
39	0.9340	0.9507	0.9090	0.9238	0.9527	1.0000	0.8750
40	0.9865	0.9826	0.9875	0.9856	0.9935	1.0000	1.0000
41	0.9282	0.9611	1.0000	0.9363	0.9487	0.9375	0.8571
42	0.9901	0.9912	1.0000	0.9866	0.9922	1.0000	1.0000
43	0.9502	0.9296	0.9500	0.9043	0.9366	0.9333	0.8000
44	0.9925	0.9912	1.0000	0.9877	0.9943	1.0000	1.0000
45	0.9404	0.9606	1.0000	0.8703	0.8584	0.9285	1.0000
46	0.9987	0.9948	1.0000	0.9954	0.9974	1.0000	1.0000
47	0.9677	0.9695	0.9444	0.9058	0.8183	1.0000	1.0000
48	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
49	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Table 9 - Probability of selecting the optimal move at any position on the board with the move suggested by HMC(n) where n is iterations

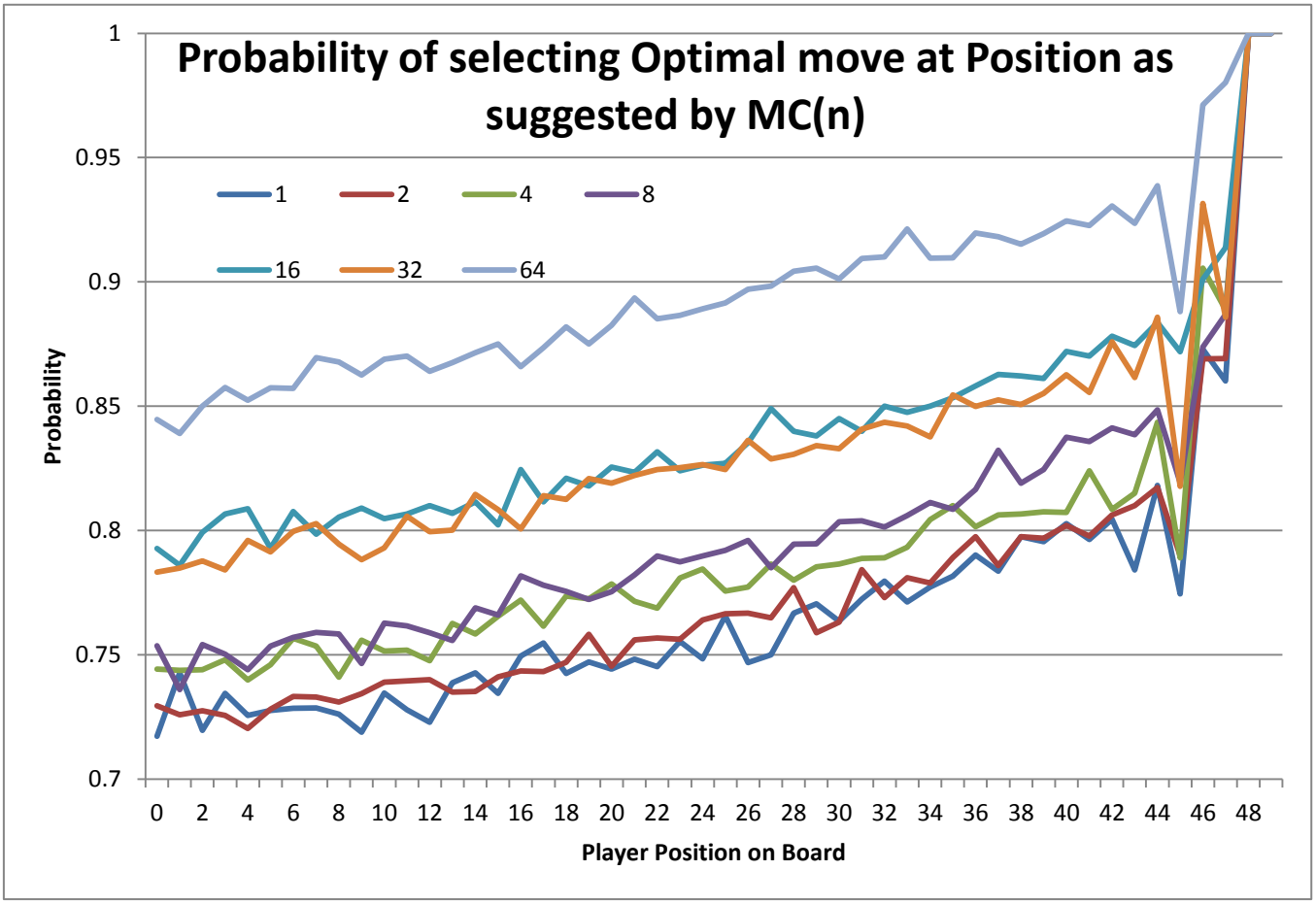

Figure 8 - Probability of selecting optimal move at Position as suggested by MC(n)

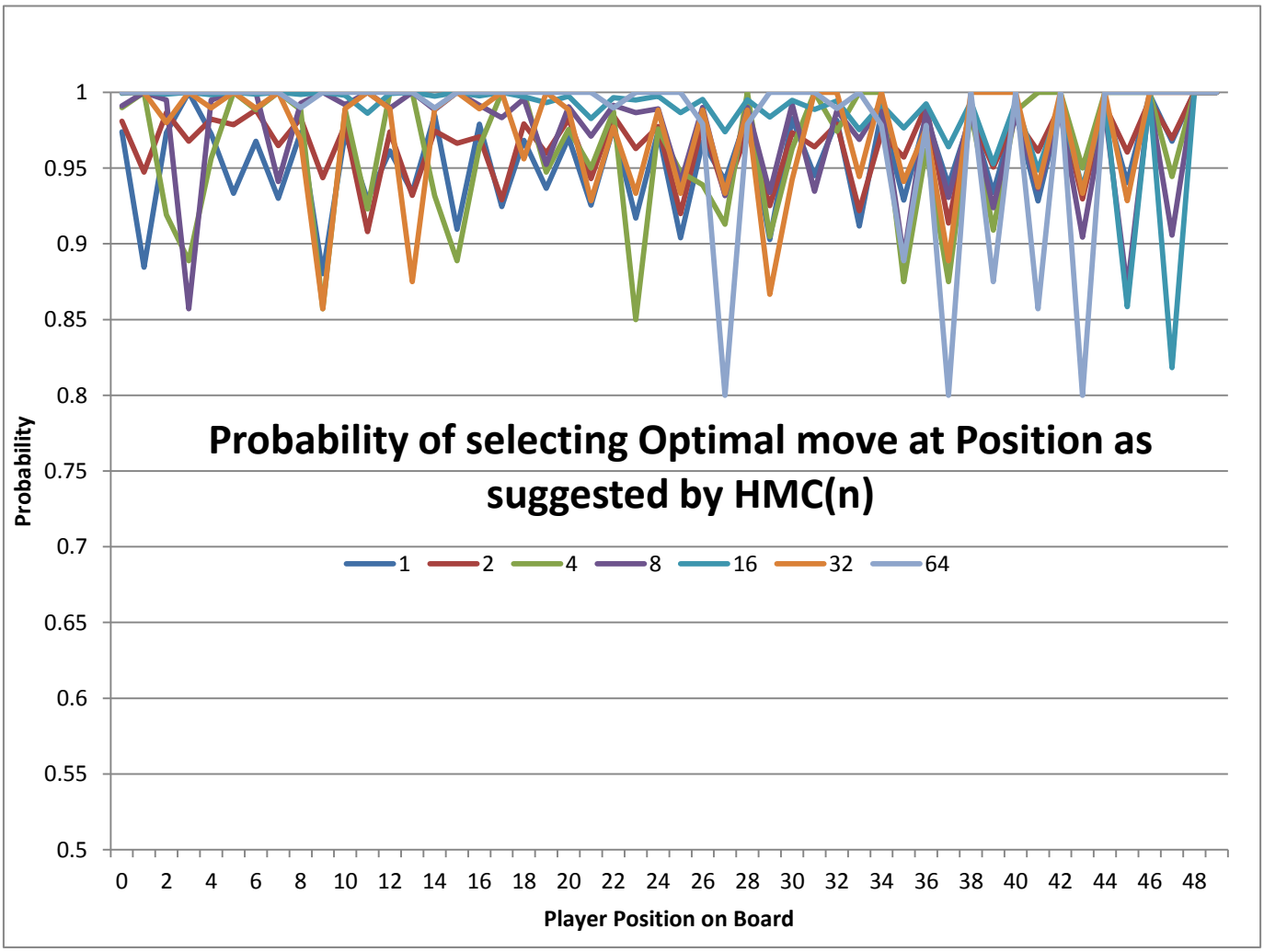

Figure 9 - Probability of selecting optimal move at Position as suggested by HMC(n)

4.3.3 Decisions suggested with regards to Oppositions Position

In this section we will again use a board size of **50** instead of 10, in order to use a larger window in which we can see the decisions made by the MC and HMC methods. This will also allow us to gain an understanding of the decision made with regards to the difference between the two players, instead of the player's actual position on the board.

Difference in Position between the Two Players	MC(n)						
	1	**2**	**4**	**8**	**16**	**32**	**64**
0	0.8073	0.7604	0.7409	0.7470	0.7761	0.8244	0.8882
1	0.8116	0.7705	0.7449	0.7517	0.7776	0.8256	0.8865
2	0.8274	0.7869	0.7564	0.7536	0.7804	0.8216	0.8913
3	0.8656	0.8159	0.7675	0.7554	0.7706	0.8119	0.8612
4	0.8988	0.8488	0.7908	0.7664	0.7711	0.8115	0.8528
5	0.8952	0.8801	0.8226	0.7834	0.7570	0.7325	0.8011
6	0.9537	0.9070	0.8538	0.8057	0.7731	0.7666	0.7876
7	0.9545	0.9469	0.8967	0.8503	0.7747	0.8200	0.6785
8	1.0000	0.9540	0.9367	0.8755	0.8052	0.7735	0.7692
9	0.0000	1.0000	0.9583	0.9416	0.8975	0.9090	0.8888
10	0.0000	0.0000	1.0000	1.0000	0.8148	0.7500	0.8000

Table 10 - Probability of selecting the optimal move depending on the difference of position between the two players with the move suggested by MC(n) where n is iterations

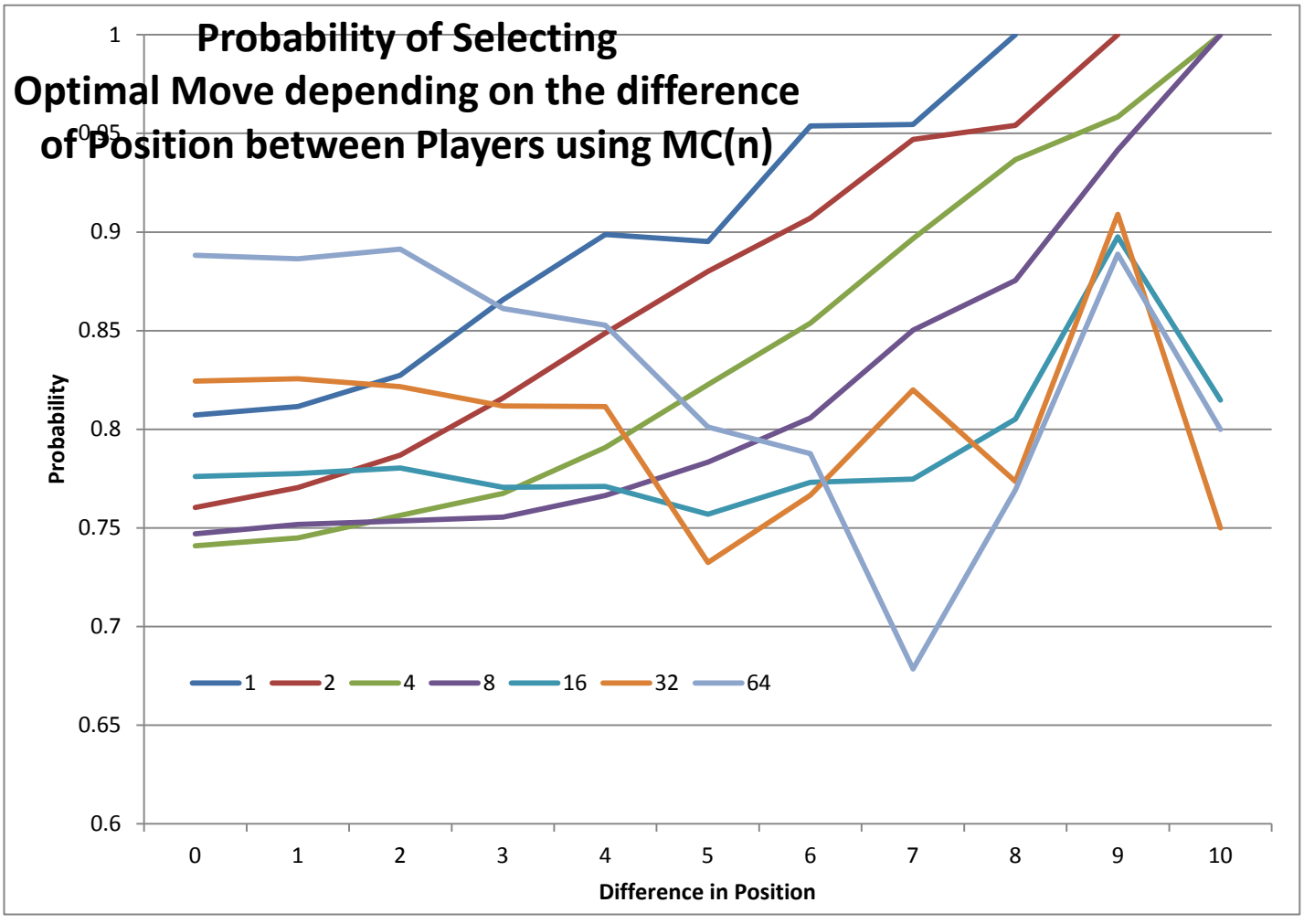

Figure 10 - Probability of Selecting Optimal Move depending on the difference of Position between Players as suggested by MC(n)

HMC(n)

Difference in Position between the Two Players	1	2	4	8	16	32	64
0	0.9955	0.9848	0.9705	0.9577	0.9769	0.9480	0.9700
1	0.9918	0.9784	0.9675	0.9493	0.9641	0.8918	0.8034
2	0.9957	0.9818	0.9721	0.9649	0.9795	1.0000	1.0000
3	0.9939	0.9818	0.9719	0.9642	0.9807	1.0000	1.0000
4	0.9872	0.9884	0.9785	0.9683	0.9813	1.0000	1.0000
5	0.9787	0.9912	0.9782	0.9610	0.9754	1.0000	1.0000
6	1.0000	0.9931	0.9814	0.9739	0.9682	1.0000	1.0000
7	1.0000	1.0000	0.9821	0.9733	0.9090	1.0000	1.0000
8	1.0000	1.0000	0.9850	0.9814	1.0000	1.0000	1.0000
9	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
10	0.0000	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Table 11 - Probability of selecting the optimal move depending on the difference of position between the two players with the move suggested by HMC(n) where n is iterations

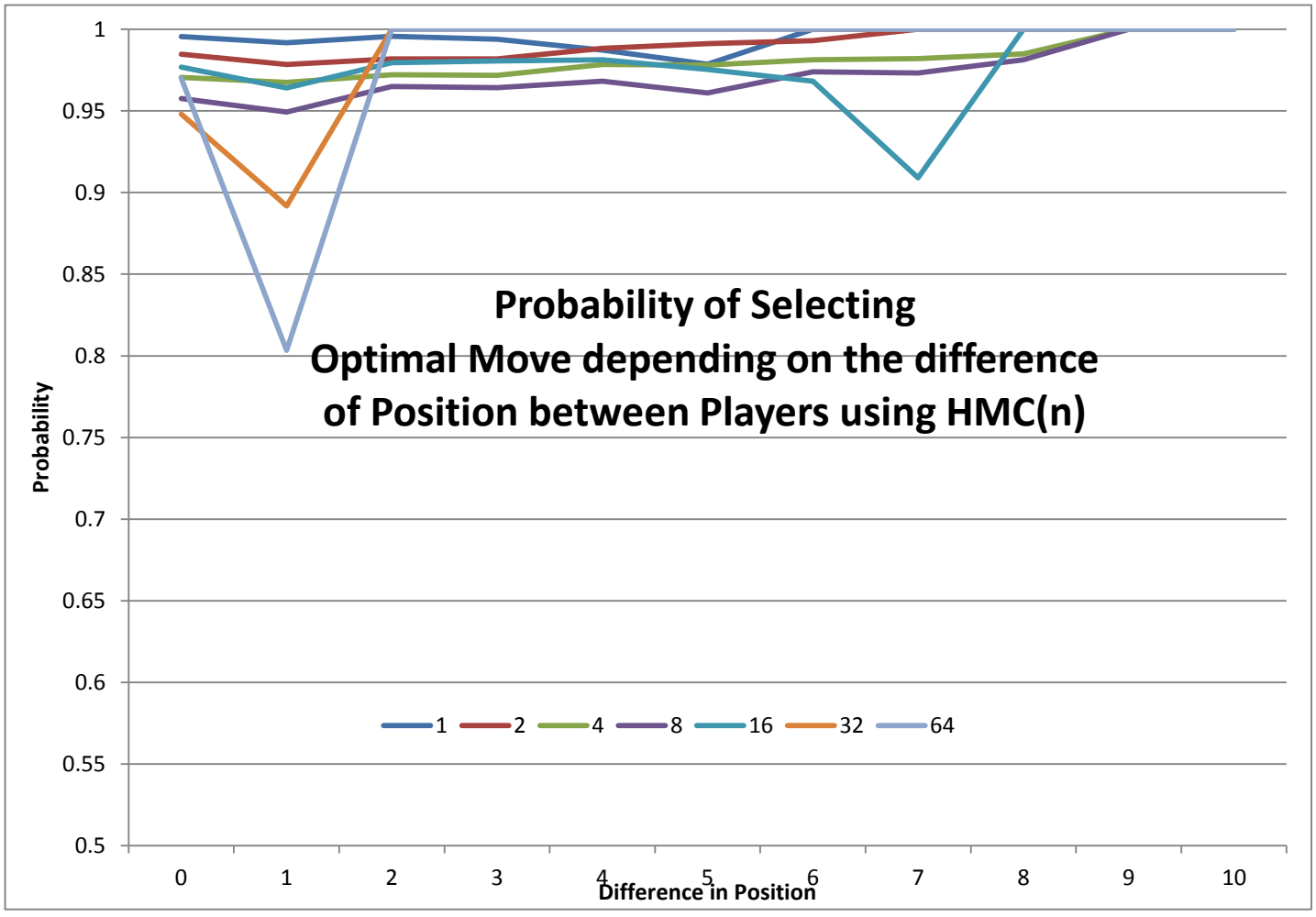

Figure 11 - Probability of Selecting Optimal Move depending on the difference of Position between Players as suggested by HMC(n)

5. Analysis

In this section I will discuss the findings from the Results section and further analyse them to see what relationships (if any) can be seen. This will include assessing the effectiveness of pitting different types of simulations against each other, understanding what effects they have on the signal-to-noise ratio, as well as seeing if any changes are made to the laziness phenomenon (Althofer, 2008) using the HMC(n) methods.

5.1 Basic Monte Carlo Patterns

Against a random player, we can see in Table 2 that the win rate for the Monte Carlo (MC) player is always above 97%, even with an MC(n) argument of only 1. This shows that even a slight sense of decision making can make a huge difference, albeit against a random player.

Looking further on, we can see that with n of value 64 we can surely defeat the computer, with a win rate of 100%. From this we can make the early statement that as the amount of Monte Carlo iterations increases, the effectiveness of the decision increases also.

On to playing against another player also using MC(n), and we can see that the above statement does hold true so far. Below in Table 12, Table 3 is conditionally formatted depending on the amount of wins for Player 1. Red relates to more wins, green relates to less wins.

Player 1

Player 2 MC(n)	1	2	4	8	16	32	64
1	6870	6881	7278	7215	7905	8780	9563
2	7261	7203	7607	7521	8117	8822	9594
4	7167	7172	7597	7529	8168	8861	9540
8	6803	6876	7358	7189	7861	8713	9506
16	6278	6338	6847	6638	7491	8452	9401
32	5566	5560	6059	6147	6896	8080	9234
64	4713	4885	5437	5314	6346	7722	9122

Table 12 – Number of wins for Player 1 (MC Player) against Player 2 (MC Player)

The top right hand corner is the brightest red in the table, indicating that the more simulations Player 1 completes, the better the decision making. The table even forms a style of upper triangular matrix, whereby the best results all exist in the top-right hand side of the table.

The inverse of this also holds, meaning that the more simulations that Player 2 generates, the better the performance of Player 2.

However, this also shows us the importance of moving first. When Player 1 (who moves first) has MC(64) to Player 2's MC(1), there is a win rate of 95.63%. Vice versa however, and there is only a 47.13% win rate for Player 2. This shows that overtaking the opponent is a difficult manoeuvre, even when using MC(n).

5.2 High Level Monte Carlo Patterns

Against a random opponent, the HMC(n) is even more effective than the MC(n), in all situations. It starts with a win rate of 99.69% at HMC(1), which is rivalling the performance of MC(64). This shows the potential of HMC(n), albeit against a random opponent.

Table 13 below shows Table 5 conditionally formatted such that the high the amount of wins for Player 1, the more red the cell, the lower the amount of wins, the more green the cell. The cell's colours are relative to the values in the table, meaning that the lowest value is the absolute darkest green, and the highest value an absolute darkest red.

Player 1

Player 2	HMC(n)						
MC(n)	**1**	**2**	**4**	**8**	**16**	**32**	**64**
1	9192	9249	9328	9580	9718	9770	10000
2	9224	9547	9504	9639	9642	9791	10000
4	9267	9366	9425	9585	9749	9795	10000
8	9087	9318	9181	9515	9650	9777	10000
16	8997	9130	9097	9398	9536	9746	10000
32	8536	8952	8864	9298	9499	9685	10000
64	8046	8733	8603	9110	9347	9579	10000

Table 13 – Number of wins for Player 1 (HMC Player) against Player 2 (MC Player)

When competing against a player using MC(n), the HMC(n) begins to show a pattern of dominance all through the values of n. When n = 1 for both methods, Table 13 shows the value of 91.92% victory rate for Player 1, whereas there is only a 68.70% win rate for Player 1 in Table 12. Even with a single iteration, the HMC(n) method proves to be a much more effective way of simulating a decision. Moreover, the performance of HMC(1) rivals that of MC(64), showing just how much of an increase simulating games on a higher level has shown. Overall, we can see that HMC(n) is effective regardless of what MC(n) is, and that when n = 64 in this experiment, it moves perfectly with every move, ensuring victory.

Even when n is low (n <= 2), the success rate is still above 80% in all situations, no matter what the value of MC(n). Winning this many games with such a small n shows the efficiency of this way of decision making overall, albeit against an opponent using only MC(n).

Another pattern we can see arising from Table 13 is the independence of HMC(n) from MC(n). Whereas in Table 12 the most effective results can be seen as an upper triangular matrix and as almost a diagonal mirror there are the worst results, the speed at which MC(n) is able to compete with HMC(n) is not great enough in order to win games without a significant boost in the amount of iterations, i.e. The minimum games won in Table 12 is equal to 4713, whereas in Table 13 it is equal to 8046.

This shows the amount of "faulty" decisions (decisions which are suboptimal but remain unpunished) that are made within MC(n), when compared to HMC(n). HMC(n) makes many fewer faulty decisions at HMC(1) than MC(1), and for it, has an increase in up to 30% in win rate.

This is further proven in Table 14, where both players are competing using HMC(n).

Player 1

Player 2 HMC(n)	1	2	4	8	16	32	64
1	8876	8926	9045	9364	9687	9706	10000
2	8694	8779	8973	9307	9675	9696	10000
4	8539	8755	8894	9236	9633	9645	10000
8	8400	8694	8747	9166	9595	9611	10000
16	8392	8662	8672	9093	9530	9599	10000
32	8156	8582	8564	9051	9439	9502	10000
64	7902	8555	8438	8991	9358	9487	10000

Table 14 – Number of wins for Player 1 (HMC Player) against Player 2 (HMC Player)

Overall when comparing Table 14 to Table 13, we can see that in Table 13 Player 1 has a slightly higher chance of winning. This change varies from around 0% to 1%. We can attribute this down to Player 2 also using HMC(n) to simulate its decisions, and therefore being able to punish Player 1 when it uses a faulty decision.

As both players are using HMC(n), the right-hand side section of the table (n >= 64) shows that Player 1 wins in all situations. This is a clear sign of perfect play, as Player 2 would also be performing the ideal movement in some cases, so the only way to win in this case is to go first.

Table 14 resembles Table 13 in its appearance, as both Players used the same method of decision making; however the difference is in the speed of change between the two tables. HMC(n) can be seen to punish a Player for making a 1 step more often than MC(n), where the play is more relaxed.

5.3 Signal-to-Noise Ratio

In Tables 8 and 9 we see the probability of either method of simulation suggesting the optimal move at that position. Table 8 provides this probability when using MC(n), and Table 9 uses HMC(n).

Figure 8 shows Table 8 in graph form, plotting the position of the Player on the board, against the probability of an optimal move being suggested. Overall, Figure 8 shows a clear relationship between the Player's position on the board, the amount of iterations that the simulation uses, and the probability of the move suggested being optimal.

In general, we can see that the probability of an optimal move being suggested increases as the amount of iterations for the simulation increases. This holds for all positions on the board, suggesting that the simulation can see a clearer picture of the end goal when more iterations come into play.

However, all lines on the graph increase at approximately the same gradient, simply being y-shifted according to the amount of iterations. This means that the signal-to-noise ratio is still relatively low in all of these simulations when the Player is at the beginning of the board. This ratio then gradually increases as the Player approaches the end of the board.

Furthermore, we can see that as the Player gets closer and closer to the end of the board, the probability of being suggested an optimal move increases, such to the point at which it becomes 1 (at position 48/49). This graph is in conjunction with what was discussed (Browne et al, 2012), in how the decisions made at the beginning of a game have less of an impact compared to those made at the end of the game; the Double Step game being more similar to the game of Go, rather than the game of Chess (where decisions at the beginning of the game can have drastic effects on the end result of the game (Ramanujan, 2010)).

Also, we see some slight nuances of laziness amongst the results of this experiment also. At values 46/47 for almost all iterations of MC(n), there is a sharp decrease in the probability of selecting an optimal move. This may be down to the simulation's assuredness in victory, causing the two decisions (i.e. moving one ahead and moving two ahead) to seem equally as likely in victory, leading to a suboptimal decision being taken.

In Figure 9 we see Table 9 in graph form, this time showing how HMC(n) deals with the signal-to-noise ratio.

As was seen in the prior section (5.2), the HMC(n) simulation requires many less iterations to be simulated in order to perform just as well as the MC(n) simulation. In Figure 9, we see that at position 0, the range of probabilities suggested (depending on the amount of

iterations) is as wide as almost 13%, whereas in Figure 9, we can see that this range is as small as 2.5%. As HMC(n) begins with an accuracy of 97.5%, this range is limited by the upper limit of 100% accuracy, or optimal play.

Figure 9 takes an interesting shape for the Player, showing almost 100% optimal moves suggested at any even value of the position, but stooping to almost 80% optimal moves suggested when it lands on an odd board position.

This can be attributed to a number of reasons. First, it could be the effectiveness of the opponent in Figure 9, which is also an HMC(n) player. This means that a single suboptimal move (at any stage in the game) causes the Player to become lazy in their decisions after that, as the simulation shows that there may not be a way to come back from that suboptimal move.

Secondly, it could simply be attributed to the fact that the sample size at these positions was not large enough. As the algorithm has proved extremely effective in its prior tests, it may have been that it only stepped onto odd squares (meaning it at some point made a suboptimal move) in a few of its simulated games, meaning that a single suboptimal decision from that position furthermore would create a large decrease in the overall probability for perfect play. Also, as the Player must land exactly on square n to finish the game, it may be that in order to correct the suboptimal decision made, another has to be made in order to return to an even square.

Overall however, it can be seen that as the number of iterations increases for HMC(n), the probability of perfect play does also increase at each position. In the case of HMC(64), it can even be said that it performs almost perfectly throughout the entire game, only suffering from the pitfalls of the odd square problem as described earlier. Even squares however, are almost always at a probability of 1 for the move suggested to be optimal.

In regards to signal-to-noise ratio, HMC(n) proves to be able to decipher the best decision much better than MC(n). As HMC(n) simulates each of the decisions on a higher level, it is able to quickly make sense of which decisions lead to victory, and which lead to defeat. This allows HMC(n) to provide a more effective form of decision making, especially earlier in the game.

5.4 Effects on Laziness

Table 7 shows a recreation of the experiment performed by Althofer (Althofer, 2008), only this time using HMC(n) instead of MC(n). The results show a much more ruthless style of game play.

In the games in which it is possible for either Player to finish before the other Player, this Player wins 100% of the time. We see this in the columns 6v3 and 6v4 for Player 2, and in 6v7, 6v8, 6v9 and 6v10 for Player 1.

Interestingly, we see a similar trend in the columns 6v5 and 6v6 for the HMC(n) Player, in that the decisions provided by the simulation are actually slightly better if the Player is slightly behind the opponent. This is shown through a smaller number of iterations being required before perfect play is achieved (32 iterations in 6v5, as opposed to 64 in 6v6).

However, we do see vast improvements even from the first iteration of the HMC(n) Player. From a 48% win rate shown in Table 1 (6v5, 1 iteration), to a 97% win rate (Table 7), this is an increase in almost 50%. Furthermore, we can extrapolate from Table 9 (HMC(n) player) showing the probability of an optimal decision being suggested even at early positions on the board being extremely high. This, when compared to Table 8 (MC(n) Player), shows why there is such a large increase in win rate.

In Figure 10 we see Table 10 in graph form, showing us the probability of selecting an optimal move using MC(n) when considering the difference in position between the two players. Note, this difference in position is absolute, meaning that the Player could be either in front or behind; the same behaviour is noted.

We see an interesting turn of events in this Figure, namely that as the difference between the two Players increases with a low amount of iterations for the simulation, the decisions suggested actually become better. When there are a large number of iterations (n > 8), as the difference between the two Players increases the decisions become probabilistically worse, and the simulation tends to suggest a suboptimal move more often.

We can explain this by understanding the simulation's assuredness in its own victory. With few iterations (n <= 8) , the simulation win rate remains near 75%. This means the game must remain closer for longer periods of time, therefore all decisions are suggested with greater "care" and less laziness is observed.

Conversely, when the iteration count exceeds 8, the win rate rises to around 90-95%. This means that the MC(n) simulation is much more assured in its victory, and both decisions (moving one forward and moving two forward) become probabilistically close. The simulation

regards both decisions as just as likely to win them the game, so more suboptimal decisions are suggested, which can account for the drop in probability for optimal suggestions.

This can also be used to explain the rise which can be seen in Figure 10, for Position 9. At position 9, at least 1 suboptimal move has had to be performed in order to be at this position. This means that the advantage of moving first no longer exists, and there instead exists pressure from the opposing Player who has caught up to the Player. This in turn creates a close situation, which in turn causes the simulation to make a better decision.

In Figure 11, we see Table 11 in graph form, showing up the probability of selecting an optimal move using HMC(n) when considering the difference in position between the two players. Again, the difference in players is the absolute difference.

Within this simulation, we see that in almost all cases, the optimal move is suggested with an accuracy of above 95%. Only on a few occasions has the optimal suggestion probability stooped below this number, and in these cases it is again suffering from the odd square problem as described in section 5.3.

The problem with MC(n)'s assuredness in its own victory seems to not appear as prevalent in HMC(n) here. When we look through the iterations in Table 11, the probability of an optimal move being suggested increases almost monotonically throughout. This shows that no real effect from the distance between the two players is felt in terms of laziness of suggesting decisions.

In actuality, when the iterations exceed 32, HMC(n) suggests perfect moves when the distance between the players is 2 squares or more. This shows the ruthlessness of the HMC(n) simulation, also giving us an insight into the value that the extra level of simulations within HMC(n) provides.

By simulating two MC(n) games instead of random games, HMC(n) is able to select the best result from two experiments, rather than two random simulations. This allows each HMC(n) decision to have the benefit of its decisions also being informed. By taking the most informed of both of the decisions, HMC(n) is able to overcome the laziness phenomenon by seeing not only how itself fares when simulated with MC(n), but also how its opponent fares using MC(n). This effectively ensures that any suboptimal move, no matter where it is performed within the game, is punished, resulting in a loss for the Player. This in turn results in the Player avoiding the suboptimal move, and instead selecting the optimal move.

This allows HMC(n) to understand the risks of a suboptimal move at any stage, and keeps the two options, both moving one ahead and moving two ahead probabilistically far apart, so that the decision to be made is clear, and is almost always (95%+) optimal.

5.5 Complexity

An MC(n) simulation is completed by creating random games of each of the deterministic situations that can arise from the current situation, expanding them, and then selecting the one that is the most effective in terms of that game. We can therefore classify the MC(n) simulation as being part of the O(n) algorithms, as the time it takes to complete is linearly relatable to the size of the input (i.e. the amount of situations that can be expanded from the current situation).

It is therefore that we can classify the HMC(n) simulation as belonging to the $O(n^2)$ algorithms. Each HMC(n) decisions creates each deterministic situation from that decision as an MC(n) simulation, which then it turn goes on to do that same but this time as random simulations. By adding this extra layer of simulations, we increase the complexity of the algorithm from O(n) to $O(n^2)$.

6. Conclusion

In this section I will summarise the findings of this project, as well as examine the inferences that this project has on the wider field. Also, we will look into what future developments can be made in this field.

6.1 Discussion

Although Monte Carlo Methods have been prevalent in games and decision making for a long time now (Brugmann, 1993), the laziness phenomenon (Althofer, 2008) has only been recently observed, and has largely been attributed to the downsides of the MC(n) simulation. There have been suggestions as to how to increase the optimality of suggested decisions using the MC(n) simulation; through implementing multiple Monte Carlo decision trees (Auger, 2011), through analysing strengths of other algorithms and implementing parts of those (Ramanujan et al., 2010) and even by implementing Upper Confidence Bounds for each decision that is suggested (Kocsis et al. 2006).

This project served to analyse how effective it would be to simulate Monte Carlo decisions on a higher level, and from that gain an understanding into why the MC(n) algorithm is slow in some states, and how this can be improved.

In the case of HMC(n) vs. MC(n), we saw a clear domination from HMC(n). In Table 5, we saw that with a single iteration of the HMC(n) algorithm, it was still able to win 80% of the games, even when MC(n) used 64 iterations. This shows its initial strength in being able to select the optimal move no matter who the opponent.

We then went on to pitting HMC(n) against itself, and saw that the results were similar to that against MC(n). With perfect play at HMC(64), we saw that there exists a limit between HMC(32) and HMC(64), in which no matter who the opponent, or how many iterations of the algorithm they use, HMC(n) provides optimal moves and causes a perfect game to be played.

When investigating the effects that the HMC(n) algorithm had on the ability to suggest optimal moves on a much larger board, we saw that there was a stark increase from the capabilities of the MC(n) algorithm. With MC(n) (Table 8 & Figure 8), we saw that as the Player got close to the end of the board, the probability of MC(n) suggesting an optimal move increased, from 70% (with 1 iteration), to the game theoretic value of 100%. HMC(n) however, remain extremely high throughout, beginning at almost 98% and ending at 100% probability of being suggested an optimal move.

We did however notice a trend whenever the Player landed on an odd square when using the HMC(n) algorithm. As the board length was an even number (50 in this case), landing on an odd square represented that any way along the line of moves that had been made a suboptimal one had been taken. This results in another suboptimal move having to be taken in order to again return to an even square (as the game can only finish on the final square, not by exceeding it). By taking this into account, we can see that in fact that the HMC(n) algorithm is not lazy, but fixes its own mistake in making a suboptimal move, and in most cases does so quickly.

Having assessed the effectiveness of the HMC(n) simulation, it was now time to recreate the experiment that demonstrated MC(n)s laziness. In contrast to that experiment (Althofer, 2008), we can see that in this case the HMC(n) algorithm is much more ruthless in its decision making. If it was possible at any stage for one Player to finish before the other, that Player always won, showing 100% win rates for either Player in each column that was not 6v5 or 6v6 (Table 7).

Moreover, we then looked at the decision making that was provided in situations depending on the distance that the Player was from the opponent. This would clearly show the relationship of laziness, as the probability of selecting an optimal decision should decrease as the Player's distance between each other increases.

Following on from what was found in other experiments (Althofer, 2008), we saw that with a low amount of iterations ($n < 8$), the Player made better decisions as the game went on, whereas when the iterations were high ($n > 8$, $n < 64$), the probability of being suggested an optimal move decreased. This can be used to confirm what was said by Althofer (Althofer, 2008), in that "the larger n, the better the MC algorithm". This meant that the opponent Player was given better decisions also, which resulted in a loss seeming more likely for the Player, with in turn led to the probability of a suboptimal move being suggested becoming higher.

When analysing the effects of the distance between the two Players when they were using the HMC(n) algorithm, we saw different results to those shown with the MC(n) algorithm. Instead of the Player becoming "lazy" at any stage of the board, the probability of an optimal decision being suggested increases both when the iterations increase, as well as when the distance between the two Players increases. We see a slight reoccurrence of the odd square problem, and the HMC(n) algorithm having to re-correct an already made suboptimal move, but apart from this, almost always the optimal decision is taken, with probabilities only ever stooping as low as 95% (disregarding odd squares).

We can attribute this to the look-ahead that HMC(n) possesses. With each decision being simulated using MC(n) games instead of random games, HMC(n) is able to surpass many of the deficiencies noted within the MC(n) algorithm, as the results it considers before making a suggestion are informed, rather than completely random.

6.2 Future Developments

Following the progressions made within this project, further areas of improvement can be identified. These can be extended from the application of the HMC(n) simulations, to the program designed for the Double Step game itself.

6.2.1 HMC(n)

The HMC(n) algorithm was successfully applied to the Double Step game within this project. This has provided a sound basis for understanding the effects it has had on the laziness phenomenon, as well as the effects when compared to MC(n). Although the Double Step game allows us to see MC(n) and HMC(n) in their most simple forms, it would provide more interesting as well as more useful statistics if HMC(n) was compared using a more competitive game, such as Chess or Go. This would allow the true extent of the HMC(n) simulation to be analysed, as well as any faults that may have not yet been observed.

6.2.2 Program

After coding the HMC(n) method and applying it to the Double Step Game, the question of depth became an apparent reality. Implementing a depth counter by which the user of the program can state how many levels of simulation they would like within the High Level Monte Carlo Methods would provide an interesting experiment; both on how this effects the suggested decisions, as well as its effect on time. Also, implementing MMTCS (Auger, 2011) using HMC(n) could be an effective way of ensuring both Player's select optimal decisions at all times, as they would have independent trees to search, but this would require a more complex test scenario. This would be an interesting development in any future work.

6.3 Final Words

The implementation of HMC(n) has shown strong gains in almost all areas that it was experimented on over MC(n). The value of determining MC(n) simulations instead of random simulations has provided a much higher level of play, decreasing what was before seen as laziness and can now be called ruthlessness. Also, we have seen that HMC(n) can effectively decipher the optimal decision with 97% accuracy through the noise of a 50-square Double Step game, as well as suggest optimal decisions (with ~95% accuracy) no matter what the gap between the two Players. Overall, this project has shown that HMC(n) can be seen as an improvement over MC(n).

7. Acknowledgements

I would like to thank my supervisor Dr. Soren Riis for his invaluable support, as well as pointing me in the right direction on a number of occasions.

8. References

Althofer, I. (2008, September 8). *althofer.de.* Retrieved October 18, 2012, from On the Laziness of Monte-Carlo Game Tree Search in Non-tight Situations: http://www.althofer.de/mc-laziness.pdf

Auger, D. (2011). Multiple Tree for Partially Observable Monte-Carlo Tree Search. *Proc. Evol. Games* , 53-62.

Baudis, P. (2011). *Balancing MCTS by Dynamically Adjusting Komi Value.* Prague: Charles University.

Brooks, S. (1998). Markov chain Monte Carlo method and its application. *The Statistician* , 69-100.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., et al. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* , Vol 4.

Brugmann, B. (1993). *Monte Carlo Go.* Technical Report.

Chaslot, G., Bakkes, S., Szita, I., & Spronck, P. (2008). Monte-Carlo Tree Search: A New Framework for Game AI. *Proceeding the Artificial Intelligence Interation Digital Entertainment Conference* , 216-217.

Chaslot, G., Winands, M., Jaap Van Den Herik, H., & Uiterwijk, J. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation* , 343-357.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. *Proceedings of the 5th international conference on Computers and Games*, (pp. 72-83). Turin.

Drake, H. B. (2010). The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. *IEEE Trans. Comp. Intell. AI Games* , Vol 2 303-309.

Drake, P. D. (2007). Move Ordering vs Heavy Playouts: Where Should Heuristics be Applied in Monte Carlo Go. *Proc. 3rd North Amer. Game-On Conf.* , 35-42.

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). *Modification of UCT with Patterns in Monte-Carlo Go.* Paris: INRIA.

Kocsis, L., Szepesv, C., & Willemson, J. (2006). *Improved Monte Carlo Search.* Estonia: University of Tartu.

Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., Rimmel, A., Teytaud, O., et al. (2009). The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computer Intelligence in AI Games* , 73-89.

MacGillivray, H. T. (1982). *Monte-Carlo simulations of galaxy systems.* Springer Netherlands.

Metropolis, N. (1987). The Beginning of the Monte Carlo Method. *Los Alamos Science* , 125-130.

Metropolis, N., & Ulam, S. (1949, September). The Monte Carlo Method. *Journal of the American Statistical Association* , pp. 335-341.

Ramanujan, R., Sabharwal, A., & Selman, B. (2010). On Adversarial Search Spaces and Sampling-Based Planning. *Proc. of the 20th International Conference on Automated Planning and Scheduling*, (pp. 242-245). Toronto.

Russell, S., & Norvig, P. (2010). *Artificial Intelligence A Modern Approach 3rd Edition.* Pearson Education.

Sheppard, B. (2002). World-championship-caliber Scrabble. *Artificial Intelligence* , pp. 241-275.

Szirmay-Kalos, L. (2008). *Monte Carlo Methods in Global Illumination - Photo-realistic Rendering with Randomization.* VDM Verlag Dr. Mueller e.K.