Computer Go and Monte Carlo Tree Search: Opening Book and Parallel Solutions

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Erik Stefan Steinmetz

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Maria Gini

May, 2016

© Erik Stefan Steinmetz 2016
ALL RIGHTS RESERVED

Acknowledgements

There are many people that have earned my gratitude for their contributions to my time in graduate school. It has been a particularly long and winding road for me, so there is a lot of appreciation to go around.

First and foremost is my advisor Maria Gini. Without her wise guidance, kindness, and patience, not to mention amazing editing skills, this path in life would not have been open to me. Maria sheparded me into graduate school and has always been there to provide encouragement, support and sympathy. For this and for being a great inspiration in life and scholarship, I express my deepest, most profound gratitude.

I would also like to thank Dan Boley who has often been a second advisor to me. His arcane knowledge of LaTeX and other computational goodies seems to know no bounds. Dan leads with his enthusiasm for all computery things, which I truly appreciate.

Thanks go to the other two members of my committee, John Carlis for his writing assistance and humor, and Michael Kac for his most excellent long-term memory, and his keen editing eye.

Partial funding for this work was provided by the National Science Foundation under Grant No. OISE-1311059 and the Japanese Society for the Promotion of Science through its Summer Program SP13052. I gratefully thank Professor Emeritus Koyama and Professor Matsumoto of the Software Engineering Lab at the Nara Institute of Science and Technology for hosting and support during the 2013 summer program, along with their continued advice and assistance.

The Minnesota Supercomputing Institute (MSI) at the University of Minnesota provided large amounts of supercomputing time and programming assistance in how to use it.

Martin Müller and Ryan Hayward at the University of Alberta have provided many

emails and conversations about the Fuego and Mohex software in addition to assistance in producing game diagrams for Hex.

On a more personal level, many friends have helped me over the years. Most notably my friend Mikey for picking up various pieces that I have dropped along the way. His assistance is always appreciated. My friend Gwynne for philosophical conversations and that never-ending touch of style. For life inspiration and leading the way into scholarship, I would like to thank both Mari Ota and Dr. Caryn Cohen. Mr Toshio Onishi and his wife Shimako I thank for being there, so kind and open-hearted.

Last, and most importantly, I will take this opportunity to express my love and gratitude to my wife Martiga and my daughter Eloise. They have given me emotional support over the many years this has taken, which has encompassed all of Eloise's life up to now. Though it may not always have been apparent at the time I appreciate this and putting up with my absence during the seemingly never-ending quest for degree.

Dedication

For showing me how to learn, for always being curious, for teaching me as a boy and befriending me as an adult, I dedicate this work to my father.

Abstract

This work examines two aspects of Monte Carlo Tree Search (MCTS), a recent invention in the field of artificial intelligence.

We propose a method to guide a Monte Carlo Tree Search in the initial moves of the game of Go. Our method matches the current state of a Go board against clusters of board configurations that are derived from a large number of games played by experts. The main advantage of this method is that it does not require an exact match of the current board, and hence is effective for a longer sequence of moves compared to traditional opening books.

We apply this method to two different open-source Go-playing programs. Our experiments show that this method, through its filtering or biasing the choice of a next move to a small subset of possible moves, improves play effectively in the initial moves of a game.

We also conduct a study of the effectiveness of various kinds of parallelization of MCTS, and add our own parallel MCTS variant. This variant introduces the notion of using multiple algorithms in the root version of parallelization. The study is conducted across two different domains: Go and Hex. Our study uses a consistent measure of performance gains in terms of winning rates against a fixed opponent and uses enough trials to provide statistically significant results.

Contents

Acknowledgements **i**

Dedication **iii**

Abstract **iv**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

2 Go and Hex Game Descriptions **4**

2.1 The game of Go 4

2.1.1 Game Rules and Play 4

2.1.2 Player Skill Levels and Handicap Play 8

2.2 The game of Hex 9

3 Monte Carlo Tree Search **12**

3.1 Tree search in games 12

3.2 Monte-Carlo Methods 15

3.2.1 Flat Monte Carlo 16

3.2.2 Exploitation versus Exploration 18

3.2.3 Monte Carlo Tree Search 19

3.2.4 MCTS Variations 21

3.3 Parallelization of MCTS 23

3.3.1 Leaf parallelization 23
3.3.2 Tree parallelization 23
3.3.3 Root parallelization 24

4 Related Work **25**
4.1 Computer Go 26
4.2 Opening Books in Go 29
4.3 Monte-Carlo Tree Search 30
4.4 Parallelization of MCTS 33
4.5 Computer Hex 36

5 Smart Start **37**
5.1 Proposed Approach 37
5.1.1 Opening books in Computer Go 38
5.1.2 Problems with Opening Books 38
5.1.3 Using Professional Play in the Opening Game to Improve MCTS 40
5.2 Design of SmartStart 41
5.2.1 Game State Representation 41
5.2.2 Database Creation 44
5.2.3 Clustering of Database Games 45
5.2.4 Finding a move during play 48
5.3 Current Open Source Go Engines 49
5.3.1 Fuego 50
5.3.2 Orego 51
5.3.3 Pachi 51
5.3.4 Gnugo 51
5.4 Results 52
5.4.1 Statistics 52
5.4.2 Experimental Parameters 53
5.4.3 Orego vs. Gnugo Results 55
5.4.4 Fuego vs. Pachi Results 55
5.5 SmartStart Summary and Future Work 56

6 Parallel MCTS **57**

6.1 Comparisons of Parallelization of MCTS 57

6.2 Multiple Algorithms for Root Parallelization 59

6.3 Methodology 59

6.3.1 Larger Trees With More Time 60

6.3.2 Larger Trees With More Threads 60

6.3.3 Using More Trees 61

6.3.4 Mixing Search Styles with Multiple Algorithms 61

6.4 Experimental Setup 62

6.4.1 Statistics 64

6.5 Results 65

6.5.1 Extended Time Results 65

6.5.2 Single Tree Multithreaded Results 66

6.5.3 Root Parallelization Results 66

6.5.4 Comparing Parallelization Techniques 68

6.5.5 Mixed Search Results 73

6.6 Analysis of Results 73

7 Conclusions and Future Work **76**

7.1 Conclusions 76

7.2 Future Work 77

7.2.1 SmartStart 78

7.2.2 Parallelization 78

References **79**

List of Tables

5.1 Fuego vs. Pachi 10. 10,000 games per tournament. 40
5.2 Win Rates of Orego vs Gnugo 55
5.3 Fuego vs. Pachi 10 with bias applied through move 12 56
6.1 Fuego vs Pachi with increasing time limits on 9×9 board 65
6.2 MoHex vs Wolve with increasing time/move limits on an 11×11 board 66
6.3 Fuego vs Pachi with increasing threads on 9×9 board 66
6.4 MoHex vs Wolve with increasing threads on an 11×11 board 67
6.5 Fuego win rates with increasing nodes on a 9×9 board 67
6.6 MoHex vs Wolve with increasing nodes on a 11×11 board 67
6.7 Fuego vs Fuego with increasing nodes on a 9×9 board 68
6.8 Fuego vs Pachi with increasing nodes on a 19×19 board 68
6.9 MoHex vs Wolve with increasing nodes on an 11×11 board 69
6.10 Fuego vs Pachi on a 9×9 board 73
6.11 Fuego no RAVE vs Pachi on a 9×9 board 73

List of Figures

2.1 Example Go boards. 5
2.2 Allowable moves 6
2.3 Ko rule 6
2.4 Handicap system: pre-determined configurations 8
2.5 Example 9×9 Hex Boards 10
2.6 Example Virtual Connections 11
3.1 Portion of game tree for tic-tac-toe game. 13
3.2 7 x 7 Go with Flat Monte-Carlo 17
3.3 MCTS Algorithm Phases 19
5.1 Board positions in the vector representation. 42
5.2 A board position in the 8 symmetric orientations. 43
5.3 Percentage of next moves found in closest cluster. 47
5.4 SmartStart Speed in playouts/s. 49
5.5 p-values for 51% vs 50% winning rates 54
6.1 Comparison of winning rates for a 9×9 board 69
6.2 Comparison of Fuego no RAVE vs Pachi 70
6.3 Comparison of winning rates for root parallelization 71
6.4 Comparison of winning rates for two Go board sizes 72
6.5 Comparison of number of choices in root parallelization 72

Chapter 1

Introduction

In the early years of the 21st century a new approach to solving search problems and playing games which are not amenable to traditional tree search algorithms has arisen. This approach uses a stochastic method called Monte Carlo search to evaluate nodes in a game or search tree. The principle of stochastic evaluation is to score a node in a tree not by using a heuristic evaluation function, but by playing a large number of test simulations using randomly chosen actions starting at the node to be evaluated out to the end of the game or simulation.

The ancient game of Go has been a grand challenge in the area of artificial intelligence for decades because it remains resistant to traditional game search techniques which have proven successful in other games such as Othello, checkers, and chess. The size of the search space along with the lack of heuristics to rate positions and moves meant that some other way would need to be found. The use of Monte Carlo methods has proven to be the key to progress in tackling Go, with a program named "AlphaGo" [1] finally achieving the milestone of defeating the top-ranked professional Go player in the world, Lee Sedol, in a highly anticipated tournament in March of 2016. AlphaGo uses a deep neural net to learn from expert human games and uses reinforcement learning to learn a policy network from self-play. Monte Carlo Tree Search is then guided by the learned probability distribution over moves and by a value network which predicts the expected outcome for positions played using the policy. Most of advancements in the methods of this dissertaion have been developed in this domain.

In this work we look at two different aspects of Monte Carlo Tree Search (MCTS).

First we improve performance at the beginning of the game of Go through a novel form of opening book which advises the Monte Carlo search. Then we examine the gains which are afforded by parallelization of the Monte Carlo Tree Search algorithm in the domain of Go and another game called Hex, along with introducing a new way to parallelize it.

We introduce SmartStart, our method which improves Monte Carlo search at the beginning of the game of Go, where its search tree is at its widest and deepest. This method uses expert knowledge to eliminate from consideration moves which have not been played by professional Go players in similar situations. We create a multi-element representation for each board position and then match that against clusters of professional games. Only those next moves which were played in games in the closest cluster are allowed to be searched by the Monte Carlo algorithm. This is a fast filter because the clusters of the professional games are calculated ahead of time. By pruning a large proportion of the options at the very beginning of a game tree, stochastic search can spend its time on the most fruitful move possibilities. Applying this technique has raised the win rate of a Monte Carlo program by a small, but statistically significant amount.

We examine the results of parallelizing the MCTS algorithm. Although increasing the amount of time to build a tree using MCTS increases the quality of the results, this may in some cases be impractical or disadvantageous. Previous studies comparing parallelization have suffered from either using too few trials to achieve a reasonable statistical significance, or have used indirect measures of the effectiveness. We compare the root and tree methods of parallelizing against increasing the amount of time available using the winning rate of the programs as the efficacy measure. These comparisons are made not only in the domain of Go, but also in the game of Hex where MCTS has been applied successfully. Our results show similarities but also some differences in how parallelization affects MCTS performance across these different domains. We also introduce a new variant of root parallelization which utilizes multiple algorithms, or multiple parameters, in order to increase the diversity of the separate trees constructed.

The contents of this thesis are arranged as follows,

- Chapter 2 describes the games of Go and Hex, which are the domains over which the algorithms will be run.

- Chapter 3 first looks at tree search as applied to games, then the ideas of Monte Carlo Tree Search (MCTS). Finally we describe the various ways of dividing up MCTS work into sections that can be executed concurrently (parallelization).
- Chapter 4 is a review of the previous work and literature on computer Go, MCTS, MCTS parallelization and computer Hex.
- Chapter 5 is our explanation of the work to incorporate a novel kind of Go opening book into a MCTS program, which we call SmartStart. We describe the shortcomings of current opening book procedures, how our procedure avoids these in addition to the details of how SmartStart works. We document how applying this has improved the winning results in tournament play against other computer programs.
- Chapter 6 describes our study and comparison of parallel MCTS techniques. We also describe a new approach to parallelizing MCTS and compare the results of this new approach to current approaches.
- Chapter 7 summarizes the conclusions from our work and describes future work in these areas.

Chapter 2

Go and Hex Game Descriptions

Both Go and Hex are games played by humans against each other as entertainment and a mental challenge. Both games are also widely used as a test of artificial intelligence techniques. The game of Go has a much longer history and is better known than Hex.

2.1 The game of Go

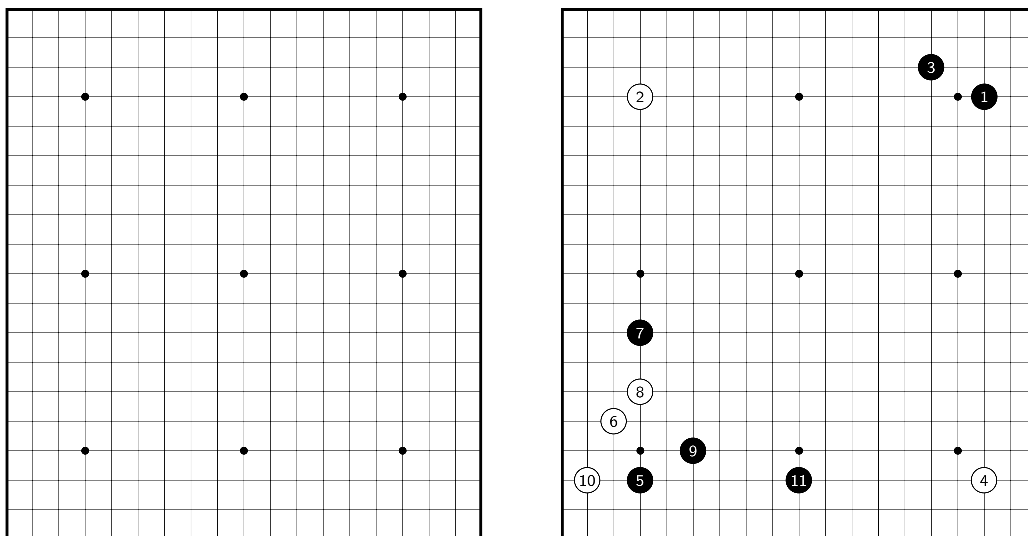
Go is a two-player, perfect information game played using black and white stones on a 19 by 19 grid of lines, with the black and white players placing single stones on alternating turns. It is called Weiqi in Chinese, Baduk in Korean, Igo or Go in Japanese.

This traditional board game has been played in its current form for many centuries. It was invented in China over 2000 years ago, having been mentioned in written texts as early as 400 B.C. Legend has it that the emperor Yao, who ruled around 2000 BC, created the game to help improve the mental acuity of his son and teach him discipline. From China it was exported to Japan by 300 AD, and also to Korea and many other parts of southeast Asia. A simple set of rules leading to very deep and difficult to understand strategies, along with the aesthetic appeal of the board and stones are some of the reasons for the game's enduring appeal.

2.1.1 Game Rules and Play

Go is played by placing black or white stones on a 19×19 grid of lines, with the black and white players alternating turns and black moving first (Figure 2.1). Once placed,

the stones are not moved around on the board as are the playing pieces in other games such as chess, checkers or backgammon, but remain in place until the end of the game, or until they are removed by capture. A player may elect not to place a stone by passing on his or her turn. The game ends when both players have passed in succession and the winner is the player who controls the most territory.



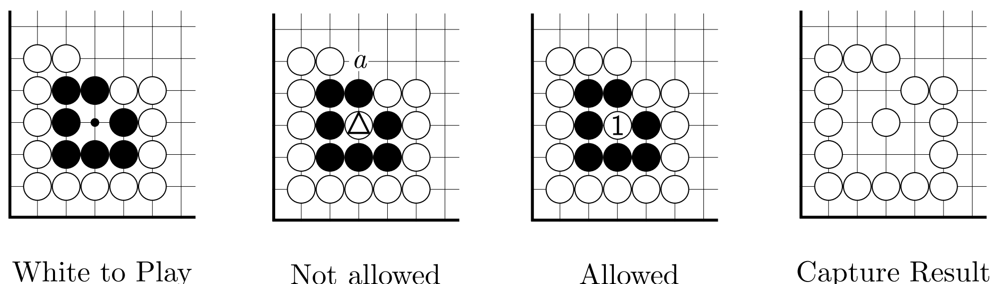
(a) Empty 19 × 19 Board. (b) Beginning Game.

Figure 2.1: Example Go boards.

The object of the game is to surround or control the territory on the board. This is done by walling off sections of the board with stones of one's own color so that the opponent cannot create a living group of stones within your controlled area. A stone or group of stones is alive, or avoiding capture, if it is connected via a line (not diagonally) to an open intersection on the board. The connection must either be direct or through one's own stones. It is thus possible to capture enemy stones by denying them access to an open intersection, also called an eye point. For example, a single stone can be captured by placing four enemy stones on each of the adjacent intersections.

Only two rules restrict the placement of a stone. The first rule is a no-suicide rule: a player may not place a stone that would cause the immediate capture of the player's stone or group of stones. Since one gauges the effects on the opponent's stones first,

however, there are many moves which capture an opponent's group which temporarily would violate this rule.

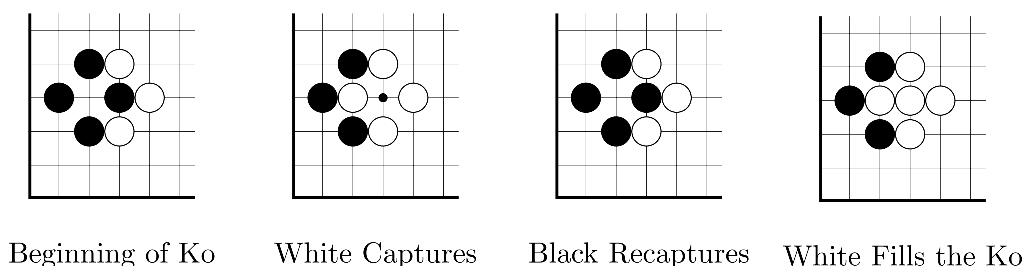


White to Play | Not allowed | Allowed | Capture Result

Figure 2.2: Allowable moves

In the situation seen in Figure 2.2, white is not allowed to play at △ because the black group still has a liberty (an open intersection) remaining at *a*. Because of this remaining liberty for black, white's play at △ would result in the white stone being immediately removed from the board. However, if the black group has no other liberties, then the move at ① would be legal as shown in the third diagram above, because it results in the capture of the black stones that surround it.

The second rule prevents the repetition of a previous position on the board. This rule is called the "ko" rule, as it prevents an infinite sequence of recaptures from a position as shown in the following diagram.



Beginning of Ko | White Captures | Black Recaptures | White Fills the Ko

Figure 2.3: Ko rule

The ko rule prevents the third move in the sequence in Figure 2.3, which would lead to an exact repetition of a board position if allowed. Black, instead of recapturing, must play elsewhere on the board (often called a ko threat), allowing white the choice

of either filling in as shown in the fourth move of Figure 2.3, or replying to black's move and allowing black to retake the ko.

When the game begins on an empty board, players place stones in strategic locations all across the board, attempting to outline areas of control, called "moyo" in Japanese. Since it takes the fewest stones to surround area in the corner of a board, initial stones are always placed first to take control of the corners, then along the sides, and finally towards the center (where it takes the most stones to surround a given amount of territory). Each player's choices affect the placement of the other player's, and the configuration of stones in one corner can have an effect on how to best place one's stones even on the opposite side of the board. Although the first moves in the game are often placed very loosely, there will be occasional sequences of stones placed adjacent to each other in a local battle for control. When they occur in a corner early in the game, the best tactics to use in particular situations have been studied enough that there are many books of move sequence collections from which people may study. These set sequences are known as "joseki" (fixed stones) in Japanese.

As the game progresses into the middle game (past the twentieth or thirtieth move, for example), the opponents engage in battles all around the board, often trading territory they may have thought of as theirs in one area, for greater influence in another. Influence is often created by building up walls of stones facing in a particular direction, and then using the wall to either surround a large area nearby, or attack an opponent's nearby moyo.

Once the board has been almost filled with stones, with both sides having recognized territories under their control, the end game consists of very short sequences of moves which define the very edges of conflicting territory, often just a point or two at a time.

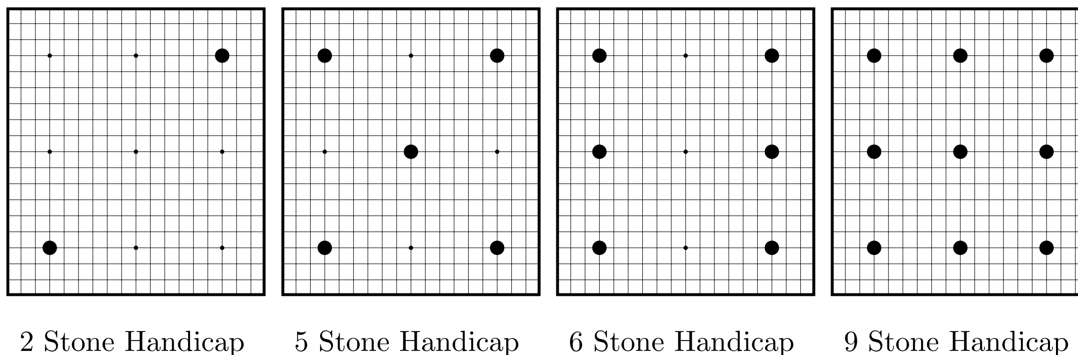
The game ends when both players have passed in succession. At that point the amount of territory controlled by each player is counted: in the Japanese system only the open territory is counted, and captured stones are subtracted from this amount. Under Chinese rules, captured stones are ignored, and all territory (stones and controlled intersections) are scored. The scoring systems produce equivalent results, with a possible one stone variance [2]. To offset the first-move advantage of the black player, a "komi" of 6.5 points or 7.5 points is granted to white: thus black must win by eight points in order to secure a 1/2 point victory when the 7.5 point komi is in effect.

In addition to playing on a 19×19 board, the game can also be played on boards of smaller sizes. Two common sizes for playing shorter, or training games, are 13×13 and 9×9. These sizes are also used to test out computer go programs.

2.1.2 Player Skill Levels and Handicap Play

At all skill levels of play, amateur players are ranked according to a system which ranges from 30 to 1 kyu, and then 1 to 7 dan. A person with a rank of 30 kyu is considered to be someone who has just learned the rules, and a 7 dan is one of the top amateur players of the game. Additionally, professional players are ranked on a separate scale from 1 to 9 dan, such that a top-ranked amateur (7 dan amateur) would be equivalent to a 1 kyu or 1 dan professional.

In order for players of different strengths to play an interesting game and have an equivalent chance of winning, Go has a handicap system which gives the black player from 2 to 9 stones placed in pre-determined spots on the board in lieu of black's first move (Figure 2.4). Typically an extra stone is given for each one or two levels of difference in the player's ranks. For example if a 2 kyu played against a 10 kyu, the 2 kyu player would take white and give a 4 to 8 stone handicap to the 10 kyu player.



2 Stone Handicap 5 Stone Handicap 6 Stone Handicap 9 Stone Handicap

Figure 2.4: Handicap system: pre-determined configurations

Among professional players, however, dan-ranked players do not play handicap games with one another. Thus many even games (games with no handicap stones placed on the board) are played between players with differing rankings, sometimes as great as 8 levels of rank.

When a handicap game is played, the stronger player, always playing white, must play more aggressively than they might against an equally ranked opponent in order to overcome the advantage of the initially placed stones for black. The white player must take risks, such as leaving a group undefended, or conducting what might normally be a risky invasion of territory, in hopes that the weaker player will not play optimally, leaving the stronger player with the advantages accrued from the bold playing style.

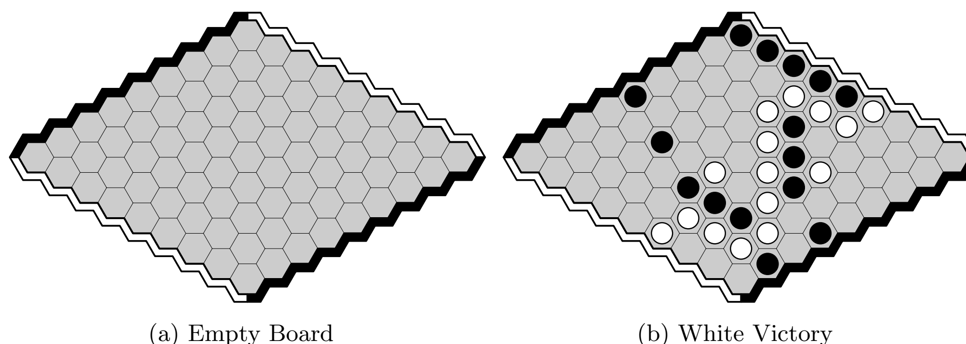
In addition to the traditional kyu and dan ranking system, a rating system similar to that commonly used in the chess world based on the work of Elo is used, especially on public go servers. The Elo system generally has ratings from 0 to about 3000 points. The point difference between two players represents the probability of winning. If two players have an equal number of Elo points, the probability for each is 50%, while a 100 point difference means the stronger player should win 64% of the time, a 200 point difference yields a 75% win rate, and so on, based on a normal distribution where the standard deviation is $200\sqrt{2}$ points. Using this system, a 7 dan amateur, or 1 dan professional would have a 2700 Elo rating, while a 1 dan amateur would be 2100, an 11 kyu would be 1000, and a beginner at 20 kyu would have a rating of 100 Elo.

2.2 The game of Hex

The game of Hex is a two-player perfect information game played by marking control of hexagonal cells on an $n \times n$ grid. The object of the game is to create a path of connected hexagons across the board, typically from top to bottom for black and left to right for white. This connection game can be played on a grid of hexagons of any size, the most common of which are 9×9 and 11×11.

The game was invented independently by both the Danish mathematician Piet Hein and John Nash in the 1940s. The game is similar to others invented at the same time such as Havannah.

The examples in figure 2.5 show a 9×9 board. The empty board is shown, and then a game won by white. The game can only be won by one of the two players, as there is no possible board configuration that would be considered a draw. Additionally, it must be won by one of the players, as there is no way to fill up the board entirely without either a black connection or a white connection occurring as shown by John Nash: there



(a) Empty Board (b) White Victory

Figure 2.5: Example 9×9 Hex Boards

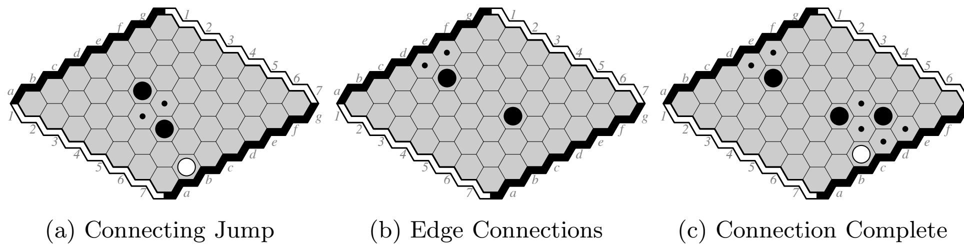
is no way to completely block one's opponent without forming a side to side connection oneself.

The player using the black pieces plays first, and can gain a significant first move advantage. Because of this, the game as played between humans usually includes a rule called the pie rule. After the first move has been placed, the second player has two options:

1. Let the move stand, play a move and continue as the second player.

2. Switch places and become the first player. The original first player now becomes the second player and plays the second move, and the game continues with the players in the swapped roles.

This rule acts as a normalizer, as the first player will typically not choose a move that is too strong since the second player would then choose to steal that move, putting the first player at a disadvantage of their own creation. In computer tournaments another solution is to have both players begin the game with fixed positions, one game for each possible starting move on the board. Thus for a 10×10 board, one game would be played for computer A beginning on hex 1-1, another for computer B beginning on hex 1-1, another for computer A beginning on 1-2, and so on for a total of 200 games: 100 for each color starting from each cell on the board. A complete set of these games is considered a single round of the tournament. Since many of the positions confer significant advantages or disadvantages, the pair of games starting from those positions will tend to be split leading to 50% win rates over many of the possible starting moves.



(a) Connecting Jump (b) Edge Connections (c) Connection Complete

Figure 2.6: Example Virtual Connections

One of the strategies in the game is making unbreakable links between cells that are two cells apart by making sure there are two connection options, as shown in Figure 2.6. This is often called a virtual connection, as a connection can be made even if the opponent moves first. In Figure 2.6(a), black's two pieces are connected with each other by a virtual connection. If white were to play in one of the dotted cells, black could occupy the other to complete the connection. In (b), black has a connection to the NW side of the board for the same reason from cell d2, but also has a connection to the SE side from cell d5 even though there are two intervening cells to the side. This is because if white were to play at c7 with an obvious blocking move for a direct virtual connection, black can play at either e6 or b6 as shown in (c), thereby completing virtual connections to both the edge and the current position at d5.

Much like Go, Hex is a game of very simple rules but complicated strategies and deep analysis must be used in order to win.

Chapter 3

Monte Carlo Tree Search

In this chapter we first take a quick look at how computers think about making choices in games using a search tree, and then describe the recent development of a variant of that process, which is known as Monte Carlo tree search.

3.1 Tree search in games

When computers need to choose an action in a game they traditionally pick the best action by constructing what is called a game tree. The tree consists of nodes, each representing a state of the game. Each node is connected to children nodes which represent the state of the game after a single action has been taken and also connected to its parent node which represents a state just before the action which led to the current state.

To choose an action, a computer will construct a tree by beginning at the current state and creating nodes representing all the possible states of the game after a possible next move and connect them as "children" of the current state. From each of these children, then, more child nodes are creating representing the possible game positions after another move, and so on. If carried out to the end of the game, this would represent a full game tree. These trees can become quite large as they represent every possible outcome beginning at the current state. For example, even the simple game of tic-tac-toe, with its decreasing number of options at each move, has a large full game tree when beginning at the empty board. There are nine children of the empty board,

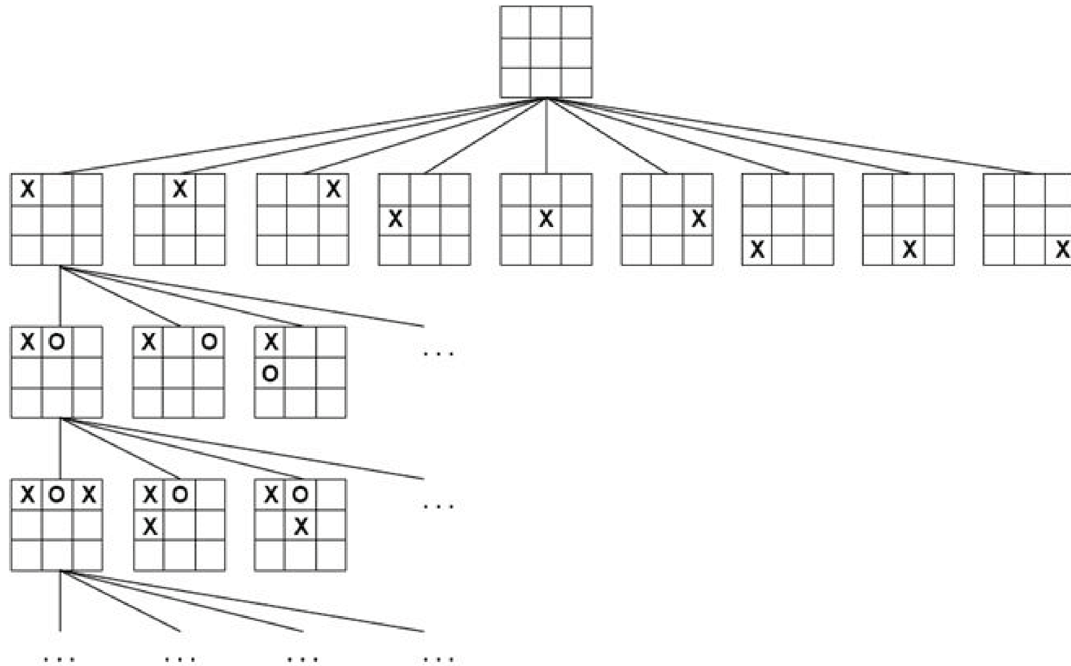


Figure 3.1: Portion of game tree for tic-tac-toe game.

eight children of each of those children, seven for each of those, and so on. At the ninth level down there are $9! = 362,880$ terminal nodes in the tree, each representing the final state of the game played out by each chosen move following the path from the root node down to that terminal node. Part of the tic-tac-toe game tree is shown in Figure 3.1. Note that there are 9! nodes in the bottom, final layer of the tree, but there are also 8! nodes in the layer above that and so on, so the total number of nodes in this tree is the sum from 1 to 9 of n!.

The number of options available from any given node is what is known as the branching factor. In the case of tic-tac-toe, the branching factor varies from 9 down to 0. In most games and their trees, the branching factor also varies, typically becoming narrower as the game progresses, but not always. For example the first move in a game of chess has exactly 20 options, but later in the game after some pawns have been cleared from the board, there are often many more options, sometimes with 14 options for just one of the pieces (a rook or bishop) alone.

There are some interesting properties to note about game trees. First of all, most

trees will not have an even depth. Even a tic-tac-toe tree is not really even, since the result of the game was determined before the ninth move in most of the games. There is no reason to actually continue expanding nodes after the result of the game has been determined, and no such continuation takes place in most game trees. Secondly, it is possible that two different nodes in the tree will describe exactly the same state of the game. These two identical states will have been arrived at through different move orderings, and so will appear in different sections of the tree, but if the current state is the full description of the game, regardless of prior move ordering, then these identical states are called transpositions, and can really be collapsed into a single node, since the sub-trees under them will be identical.

Once a tree has been built, it is used to determine the computer's next move. In order to do this, the computer must determine which of the immediately following nodes, those at the first level, is the best.

If the tree is completely filled out so that each path ends at a completed game (which is possible for games as small as tic-tac-toe) then each terminal node, also called a leaf node, that is a win for the computer player would be given a score of 1, each loss a score of zero and each draw a score of 0.5. Using a procedure called min-max, we can proceed from the bottom up assigning scores to non-leaf nodes in the following fashion. If the child nodes represent the choice of the the opponent the parent node must be given the minimum score of all its children, since we assume that the opponent will play the best move for it and thus the worst for the computer. If the child nodes represent the choice of the computer, then the parent node will be given the maximum score available from its children, since the computer is allowed to choose the best option available to it. This process allows us to fill in the values moving up the tree until finally at the very top level, the computer picks the maximum of its immediate children as the next move. In this fashion the computer will be able to pick the best move at each point in the game, and achieve what is known as perfect play.

Most of the time, however, it is not possible to create a full tree due to the branching factor of the game and the depth of the tree. When this is the case, it is still possible to create a partial game tree to build an approximation of the best move. This is done by using a static evaluation function to give an approximate value to a game state which is not a final, or end of game state. This static evaluation function allows the computer to

distinguish game states which are better or worse for it without having to read forward to the end of the game. The computer can build out the tree to chosen depth, and then use the static evaluation function on all the leaf nodes, and then as before use the min-max algorithm to propagate these values up the tree, allowing the choice amongst the next available moves from the root (current position) of the tree. The closer the static evaluation function is to the true value of the game state, the better this algorithm will function. Indeed, if there were a perfect evaluation function available for a game, one would not have to build a tree at all: simply expand the current state to all possible successor states, apply the function to each of those states, and then pick the option with the highest function value.

Since game trees can become very large, computer algorithms often will eliminate parts of the tree from consideration if it would be impossible or unlikely to find a solution in that branch of the tree. This process is called pruning the tree. One improvement on the min-max algorithm, called alpha-beta ($\alpha - \beta$) [3], eliminates from the tree some sub-trees that cannot produce better results than their siblings by running the evaluation function at each level and then not pursuing those sub-trees where a sibling node has already shown that it can achieve a better result. This pruning allows the computer to spend its time doing evaluations in more promising parts of the tree at greater depths, where the evaluations should be more accurate. The efficiency of $\alpha - \beta$ depends on the order in which subtrees are evaluated, however, since the more promising subtree has to have been discovered first in order for a later less promising one to be pruned.

3.2 Monte-Carlo Methods

When a search tree is too large, or a reasonably fast and accurate static evaluation function is not available, traditional min-max search will fail to deliver good results. In these cases another way to get an evaluation of the node is to use stochastic sampling. For the game of Go this was first proposed by Brügmann in 1993 [4], and completes games with random move sequences to the end of the game many thousands of times from a given position.

3.2.1 Flat Monte Carlo

The basic idea of stochastic sampling methods (named Monte Carlo for the gambling reference) is to assign a value to a game state by playing random legal moves starting from the game state in question until the end of the game where a win or loss can be determined. These move sequences to a terminal state are called a playout. Because this process tends to be rather fast, a large number of playouts can be executed for each game state that needs to be evaluated. For example, executing one thousand playouts (that is, one thousand independent randomly played games to completion) from each node that needs an evaluation. Each of these nodes then will have a score based on how many of the playouts resulted in victories for the computer player divided by the number of playouts from that node. Alternatively the score can be based on the amount of the win or loss, if that information is available in the game [5]. For example in Go the game can be won or lost by a certain number of points, while in Hex the result is simply a win or a loss. If the playouts are distributed in an even fashion across all the possible actions for the computer, this is called flat Monte-Carlo search. As an example imagine our tic-tac-toe game again. For each of the nine possible moves we would play one thousand games with random choices until the end of the game. We thus have a random sampling of the possible results in each of the nine sub-trees, and we can pick whichever of these samplings shows us the best result. In our small tic-tac-toe game this is a fairly large sampling (1000 out of about 40000 possible games in each of the nine branches). As the trees get larger, however, the sample size gets proportionally smaller, and does so quite quickly. Even on a very small Go board of 7×7, there are 48 options for move 2, 47 for move 3 and so on. Though not all will be legal, there will be therefore on the order of $47! = 2.5 \times 10^{59}$ possible games, so our 1000 games will be a very small sampling indeed.

Flat Monte Carlo evaluation does produce results, but is handicapped by a number of shortcomings. Because so many playouts are spent on exceptionally suboptimal moves, it does not scale well. Additionally, there are situations in which an incorrect move will be more likely chosen even as the number of playouts increases, due to the lack of an opponent model [6].

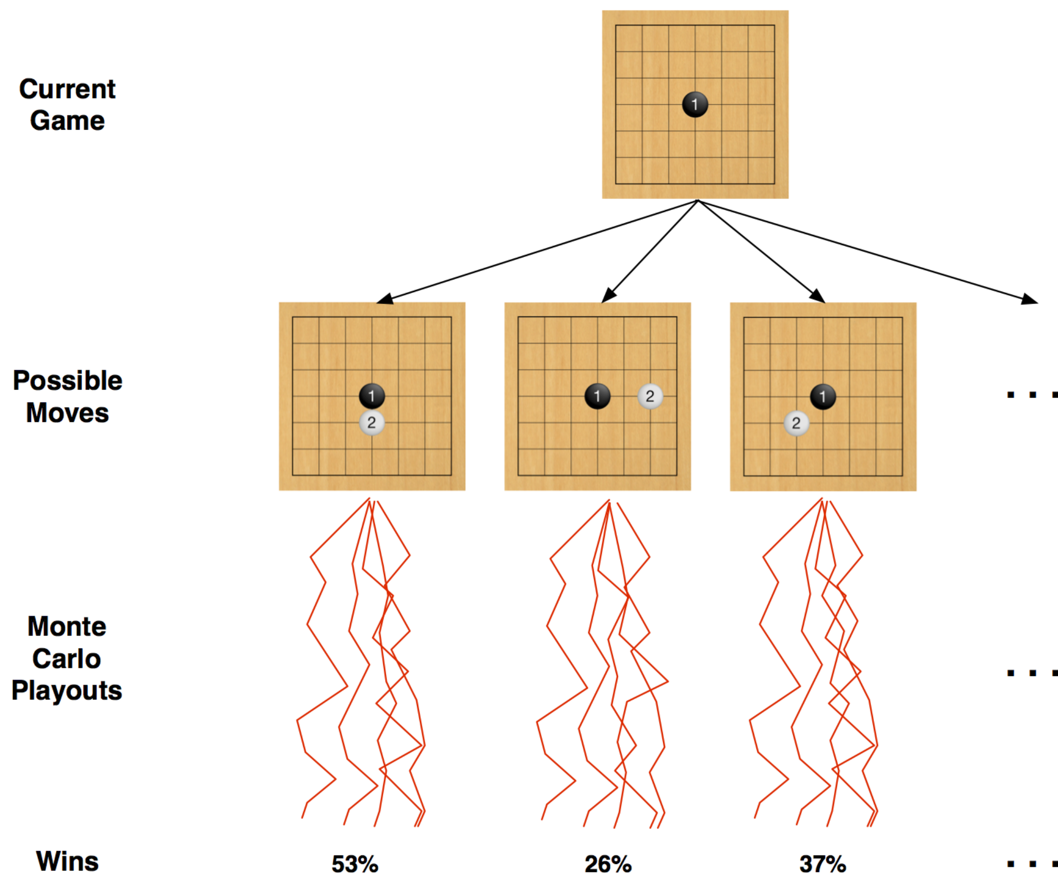


Figure 3.2: 7 x 7 Go with Flat Monte-Carlo

3.2.2 Exploitation versus Exploration

We are often constrained by a budget of time which can usually be directly translated into a certain number of playouts. So in order to improve upon flat Monte Carlo, some ideas are used from game theory, specifically the problem domain called the multi-armed bandit problem. This problem comes from the idea of playing many slot machines (one-armed bandits) which have different payout ratios, but the only way to learn the payout ratios is to play the machines. With a limited number of coins to put into the machines, one would like to balance playing coins in the machines with already-discovered good payout ratios (exploiting the knowledge of payouts) and playing coins in machines which are relatively unknown (exploring). This has been formalized in terms of minimizing a player's regret, defined the expected loss due to not playing the machine with the best payout ratio with every coin.

In terms of choosing playouts in search of the best move, the idea is to spend more of the playouts to get a better sampling of the more promising options and spend fewer playouts on the less promising options. A way to choose the option that minimized regret was proposed by Auer et al. [7], called UCB1.

$$UCB1 = \bar{X}_j + \sqrt{\frac{2\ln n}{n_j}} \tag{3.1}$$

This is called the upper confidence bound and indicates the notion that a given choice will be the optimal choice, without there being any prior knowledge of the value of a choice. The first term $\bar{X}_j$ represents the average winning rate discovered so far for the given option j. This is called the exploitation term. The rewards for each option are in the range [0,1] so this average term is also in that range. The second term $\sqrt{\frac{2\ln n}{n_j}}$ gives value to those options which have not been explored as much as the other options: this value gets larger as the proportion of the overall budget on it decreases. This is the exploration term. In order to choose which move to sample with the next playout one picks the choice which optimizes this UCB1 value. As the number of samples increases, it has been shown to asymptotically approach the true value of the node. Options which have not been explored yet have an exploration term of infinity, so each option will be explored with at least one playout.

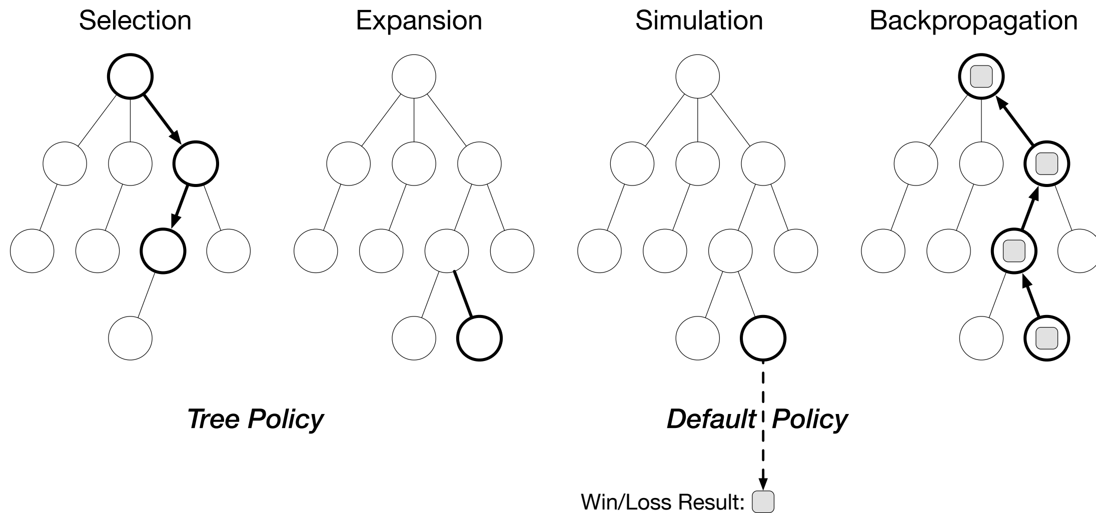


Figure 3.3: MCTS Algorithm Phases

3.2.3 Monte Carlo Tree Search

In order to make use of this decision-making policy, instead of just creating only nodes for each of the current choices, that is for each child of the root node, one builds up a tree of nodes. Doing this adds a single node to the tree with each playout executed. From the root node a node will be created for each option from the root position and a playout will be run through the created node. Once each option from root has been covered once and a child node created for each of those options, a playout will descend through one of the nodes. The choice of node is often based on the one with the highest score from the UCB1 equation above, and a second-level node will be created as a child of the chosen node on the first level. Each time a first level node is chosen, one of its child options will acquire a tree node until all of its children are filled in before third level children are created from that second level node. A tree is thus built in an asymmetric fashion, with the more promising parts of the tree expanded to deeper levels than the less promising parts.

As this tree gets created the algorithm follows these basic steps until a resource limit (such as time or number of iterations) has been reached: Selection, Expansion, Simulation, and Backpropagation.

The selection phase starts at root, choosing the action or move represented by a

node in the tree according to some selection policy. It then recursively descends the tree always choosing a child node according to the selection policy. When it reaches a node which has unvisited children and which represents a non-terminal state, the selection phase stops.

The expansion phases now creates a new node as a child node of the selected node and adds it to the tree.

Now the third phase begins: simulation plays legal actions (moves) randomly until an end position is reached. The win or loss from this simulation, also called a playout, is determined.

This is added to the statistics of the newly created node and all of its parents up to the root node during the backpropagation phase.

These steps are shown in Figure 3.3.

The choice of child nodes in the selection and expansion phases is determined by what is called a tree or in-tree policy. This is often something similar the UCB1 equation, called UCT:

$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}} \tag{3.2}$$

where n is the number of times the current (parent) node has been visited, n_j is the number of times the child j has been visited and C_p is a constant. When n_j is zero the UCT value for that child will be ∞ which will guarantee that each child of a parent node will be explored at least once before any of the children are expanded. Although all children of an expanded node must be run once before any of them can be expanded, the growth of an MCTS tree is not similar to breadth-first search. It is asymmetrical, so that the more promising parts of the tree are explored in greater depth first.

The choice of actions during the playout, the simulation phase, is called the default policy or sometimes the out-of-tree policy. It is usually a policy of picking random actions which are legal to take from the given state.

Both the tree policy and the default policy can be modified in order to alter and improve the way MCTS works, often depending on the domain to which it is being applied.

An interesting property of MCTS is that it operates as an anytime algorithm. After each iteration the scores are back-propagated up the tree so that the values for the

choice of the next move is always up to date. As the number of simulations grows towards infinity, it has been shown that the likelihood of selecting the incorrect action converges to zero [8].

The MCTS method is also interesting in that it can choose actions without utilizing much domain-specific knowledge. Any domain which can be modelled as states with allowed actions can be searched as an MCTS tree using just the rules of the game. This is in contrast to the minimax search which requires a domain-specific heuristic to perform a static evaluation of non-terminal nodes.

3.2.4 MCTS Variations

Improvements and adaptations of the MCTS algorithm are usually applied and tuned based on the domain. Most adaptations fall into one of two categories: modifying the in-tree selection policy or changing the out-of-tree default policy. Because modifying the random pick of the default policy often slows down the rate at which playouts can be executed, these are often called heavy playouts.

One major advance that is applicable in some domains is to make use of what is called the All Moves As First (AMAF) property. In many situations a move played later in the game is also valuable in the current situation. To utilize this property a separate set of statistics is kept about wins and losses in each of the nodes in the tree. The values are accumulated by looking at the actions taken in the playouts. During each random playout (using the default policy outside of the tree), every move taken is noted along with the result of the playout. If any of these moves could have been made inside the tree during the selection phase that led to the playout, the nodes resulting from those moves will have their AMAF statistics updated with the win or loss. The selection phase is then modified by using the policy that considers both the UCT score and the AMAF statistics. For example the α-AMAF algorithm gives an in-tree value to a node of

$$\alpha A + (1 - \alpha)U \tag{3.3}$$

where U is the traditional UCT value.

The most popular version of AMAF is called the Rapid Action Value Estimation (RAVE) formula. In RAVE the value of α in the α-AMAF formula is not a static, but

instead changes with the number of visits to the node, linearly decreasing towards zero at some number of visits V. Once α becomes zero the UCT score reverts to its original form.

The effect of using an AMAF algorithm is to quickly build up some statistics for child actions that have not otherwise had a playout run from them. This rapid accumulation of knowledge, though less accurate than the pure UCT knowledge, has been shown to increase the performance of MCTS in a number of domains, including Go.

A modification of the tree policy which encourages early exploitation of promising nodes is to assign a first play urgency value to unexplored child nodes instead of the standard infinite value. By doing so it is possible to explore a child node (expand it) before having to run at least one simulation through all of its siblings. Tuning this value enables a promising branch to be explored quickly, at least until the value is low enough to be less promising than the first play urgency value.

Another tree policy modification involves using domain dependent knowledge to bias selection in the tree by modifying the initial statistics associated with a node when it is built. When a node is created, its values for the number of times visited and the number of wins are set to zero by default. In order to bias a parent's selection of a node, these values can be set to non-zero values. For example, setting the values to 20 wins out of 20 visits would typically cause the parent node to want to exploit such a favorable node. Using whatever domain knowledge is available an algorithm can set such a bias on nodes representing preferred game states.

A popular way to modify the out-of-tree (default) policy from a random pick of available actions to something more realistic is to use some pattern matching. When part of the current state during a playout matches one of the available patterns, the next move in the playout is chosen from the matching pattern's suggestion rather than as a random pick. In Go small patterns, sometimes only 3×3, have been used to modify playouts based on very local situations. Similarly in Hex, patterns that choose a move which defends a virtual connection have made playouts more realistic and the overall algorithm more successful.

Another default policy modification involves a simple heuristic in the playout which stores for each player the most recent winning reply move to a given move [9]. In the playout phase, instead of a totally random move, a lookup is done to see if a move has

been played following the immediately preceding move and if that move led to a win for the currently moving player. Because this involves only 361 possibilities for each player, the lookup table is small and fast, and does not substantially slow down the playout speed. If a move is not found in this table, the normal default policy is used. This policy was also expanded to include a table based on the two previous moves, and a policy involving “forgetting”, which removes losing picks from the lookup table [10].

3.3 Parallelization of MCTS

The nature of MCTS, with its repeated and rapid playouts along with separable steps in the algorithm, afford a number of opportunities to spread the work out among parallel processes. There are generally three ways to parallelize search in an MCTS tree: at the leaves of the search tree, throughout the entire tree, and at its root.

3.3.1 Leaf parallelization

In leaf parallelization, after a node is selected in the tree for expansion, multiple playouts originating at that node are conducted in parallel. When all the playouts have finished, the combined score from those playouts is propagated back up the tree. Depending on the number of parallel threads this can greatly increase the speed at which the win-rate statistics are gathered. However, since each playout may take a different length of time, some efficiency is lost waiting for the longest playout to complete before reporting the statistics. Some implementations have moved these playouts to a GPU [11]. Unfortunately many of the playouts in leaf level parallelization are wasted due to their simultaneous nature: if the first eight playouts all are losses or very low-scoring, for example, it is unlikely that the next eight will do any better, leading to an inherent limitation of this technique.

3.3.2 Tree parallelization

In tree parallelization, multiple threads perform all four phases of MCTS (descend through the search tree, add nodes, conduct playouts and propagate statistics) at the same time. In order to prevent data corruption from simultaneous memory access, mutexes (locks) are placed on nodes that are currently in use by a thread. These locks

lead to scalability problems since many threads may want to be exploiting a particular node in a tree. One solution to this mutex problem has been to introduce virtual losses. When a thread descends through a node in the tree, it adds a virtual loss to the statistics of that node, making it less appealing to other threads, and so less likely that these other threads will try to descend through that node and be delayed by the lock. When the thread updates statistics, it removes the virtual loss. Another implementation [12] created a lock-free version of tree parallelization utilizing the virtual loss system. Without locks this method depends on a large enough virtual loss to deter other threads to the point that overwriting of old data will occur very rarely. Tree parallelization can be created on shared memory systems or on distributed systems with very fast interconnects (clusters).

3.3.3 Root parallelization

In root parallelization, multiple independent trees are built by separate threads with no information communicated between them while the trees are being built. This can happen either on a shared memory machine or in a cluster. All the trees are created through the end of the time limit on the machine or machines, and then the scores for the top layer nodes, that is to say the nodes which represent the immediate choice from the root node, are combined to determine which action will be chosen. Although the trees have been created starting from the same node, the stochastic element of MCTS means that each tree will be formed differently from the others. When the information from the different trees is combined, two methods of combining the values from the trees are commonly used. The first is to add up all the scores for each possible action from all the trees. In this case the combined score for an action is the sum of each tree's score for that action. The second method is to choose the action which "won" the contest in each of the trees. These two methods are called "average" voting and "majority" voting respectively.

Chapter 4

Related Work

Most game-playing programs depend on building and searching a game tree, using a position (or node) evaluation function and a variation of the minimax algorithm called $\alpha - \beta$ search (originally by John McCarthy, first described in [3] and [13]). This search technique looks at all the possible moves from the current position, and then each of the opponent's possible moves from each of these positions, and so on, building up a tree of possible games. Each layer in the tree adds one more move to each of the possible games. In order to find the best friendly move, $\alpha - \beta$ search looks at the best move of the opponent given a friendly move, and uses the score of that position (the result of the friendly move followed by the opponent's best move) as the assumed score of that move. Assigning a score calculated in this fashion to each of the possible friendly moves allows a game engine to choose a move.

In small games without many possible moves, for example tic-tac-toe, it is possible to fill out an entire game tree to the end, looking at every possible game sequence and its conclusion, and so determine the best move. In any non-trivial game however, the exponential explosion of the number of possible game sequences quickly makes building a full game tree infeasible. Therefore most game engines play out only a certain number of moves, and then use some heuristic to judge the value of a non-terminal position in the game. This heuristic is called an evaluation function. Many algorithms are also used to prune a game tree, throwing away certain possible sequences after only a few moves in order to be able to spend the computational time expanding other branches of the game tree (those with greater possibilities) to a deeper level.

The complexity of Go may be compared to that of chess, in that examining the game tree of either is a non-trivial exponential problem. But the exponential factor of go is much higher than that of chess. Where the average number of choices for a move in chess is 20, the average in go is about 200, and while the average length of a game of chess is on the order of 80 moves, in go the average length of a game is over 200 [14]. To look at the game tree out to four moves in chess, therefore, would require approximately 20^4 nodes (160,000), while to look ahead the same number of moves in a go game tree would require 200^4 nodes (1,600,000,000). Adding just two moves (one move for each player) bumps these numbers up to 64 million for chess and 64 trillion for Go.

In addition to the large difference in the expansion rates of the game trees for Go versus chess, it is also far more difficult to produce a static evaluation function. In chess, even counting the number of and power of the pieces on the board can be a good beginning at evaluating the board at any given time, while in Go, it can be a difficult problem simply to determine if some of the pieces on the board are to be considered "alive" or "dead" (pieces that are in an untenable position remain on the board until the opponent actually captures them and removes them from the board), and so difficult to gauge the amount of territory controlled by either side. For example, it is often the case that a small, local tree search must be run just to determine the life or death status of a group before being able to include it in a scoring of the overall board [15].

In chess, using a traditional tree search, the Deep Blue machine which beat Kasparov in the 90s would evaluate many millions of board positions per second in order to expand the game tree to a deep enough level to produce its master-level play. The best Go programs of that time were not able to give a score to even tens of thousands of game positions per second due to the lack of a fast and accurate static-board-evaluation algorithm.

4.1 Computer Go

There has been a large number of efforts over the years to create a computer Go playing program which can play at the level of a professional Go player. Up until about 2005 programs improved slowly using variations of traditional techniques and reached approximately the level of a medium-strength beginner.

In an attempt to overcome the difficulties mentioned above, the architecture of pre-MCTS Go programs (see surveys by [16] and [17], along with [18]), and of an open-source effort, Gnu-Go [19], were quite different from that of the traditional full tree search programs. Although early attempts at Go programs such as [20] used traditional methods combined with sophisticated algorithms for pruning the large size of the tree, most Go program architectures instead contained a number of different move-generating modules, an idea first used in [21]. Each module produces a suggested move along with an urgency, or importance, score. From the list suggested, the main program thread chooses one move given the importance scores and a current global state. Modules are often based on particular goals of the game, or on particular sections of the game. For example, a life and death module which produces moves to save or kill a group of stones, an opening module to produce moves which attempt to form "moyo" or frameworks on the board, a connection module which produces moves to connect friendly live groups, or an endgame module which can read and play for the most points when the board is almost filled. Additionally, some work has gone in to automating the creation of the rules used by these modules to suggest moves [22]. A combination of modules and traditional search, sometimes used to play out tactical situations, was introduced with [23] and became the standard for most serious go programs. Programs of this type, including the introduction of parallelization methods [24], made slow progress from the 1980s to the 2000s, but none ever broke through to dan-level play.

The paper "Life in the Game of Go" [25] helped give an algorithmic definition of a live group in the game, and explored life and death problems.

A comparison of chess versus Go, including their applicability as research topics and their difficulty for computers is found in [26]. Burmeister and Wiles [27] then looked at creating infuence maps based on people's perceptions of the Go board. This was then used as an example of relating local to global factors in thinking and cognition in [28]. They also did a study [29] of people's memory of board positions. Burmeister then used these works to inform a survey [30] of how Go problems can be used in the study of cognitive science, including looking at memory and problem solving.

Bouzy [31] argues for the need of spatial reasoning in order to analyze Go positions. This sort of reasoning is carried forward in [32] applying mathematical morphology, recognition of shapes, to computer Go.

Bouzy and Cazenave [33] look at the shared aspects of complex problems alongside computer Go. They cite economic, social sciences, war simulations, linguistics, and earth sciences.

Using the idea of thermography to analyze Ko situations is explored by Berlekamp [34] and also by Müller, Berlekamp, and Spight [35]. Additionally the value of multiple Ko situations is studied in [36] and [37]. These ideas are then further expanded in an "enriched environment" and applied to Go endgames and Amazons in [38]. Silver, Sutton, and Müller [39] also use these ideas under the name of "Temporal Difference Search" and Kao [40] explores applying Temperature Search to endgames.

Cazenave [22] created a system to automatically learn local patterns and positions as tactical rules. These rules were used and developed in his "Gogol" program, a rule-based computer Go program that reasons about achieving small local goals.

Bewersdorff [41] studies end game and life and death situations in terms of combinatorial math. Chen and Chen [42] also provide some mathematical definitions of life and death in Go. Wolfe [43] provides a proof that the endgame of Go is P-SPACE hard.

Huima [44] explains the Zobrist hash function which is used to hash board positions for quick lookup and storage in a table.

In 2000 Graepel [45] applies common fate graphs and subgraphs to 9×9 Go while Wolf [46] presents a study of using heuristic search techniques to solve live and death problems. These techniques are later expanded and applied in [47].

Bouzy and Cazenave [16] authored a very extensive survey of computer Go in 2001 along with Chen [48] who provides an overview of the architectures of most of the programs available at that time. Cant et al. [49] looked at some these same programs and tried to add an advising neural net into a go-playing architecture.

Bouzy [50] explores the general decision-making process of Indigo, a typical Go engine of the pre-MCTS era with a detailed overview of the modules involved.

In 2005 Chen [51] proposed a non-territory way of scoring go game positions, based on a chance-of-winning value. Nakamura [52] also worked on a static evaluation function of Go, incrementally updating information from turn to turn.

Van der Werf et al. [53] present a learning system to predict life and death situations in Go. They develop a classifier based on various features of blocks of stones including number of liberties, the perimeter, player to move, whether or not a ko exists, etc.

Nijhuis [54] also develops a system to learn Go board patterns with common fate graphs and relative subgraphs.

Wu and Baldi [55] use a DAG recursive neural network to try and learn a static evaluation function for 9x9, 13x13, and 19x19 Go boards. They use 3x3 patterns from the games as inputs to the RNN. Bouzy and Chaslot [56] also worked on automatically extracting various sized patterns from 19×19 games.

Stern et al. [57] use Bayesian patterns rankings in order to predict moves in Go games. They harvest various sized patterns from professional games based on the stones surrounding the most recent move. In 2007, Silver, Sutton, and Müller [58] used reinforcement learning to develop small patterns for helping play on Go boards up to 9x9.

Mayer [59] examines how a board is represented while training neural networks with temporal difference learning. This is done on a very small (5x5) board.

Lee et al. [60] build an ontology of computer Go knowledge using expert input.

Wistuba et al. [61] compare different Bayesian move prediction systems in 2012 while Wolf [62] looked at techniques for recognizing and moving in Seki in Go.

Maddison et al. [63] use a DCNN trained on a database of professional games to predict moves in games. When used without search it beat GnuGo and matched Fuego at 100k playouts/turn.

4.2 Opening Books in Go

An opening book is a set of predefined moves that can be found by matching the state of the board at the beginning of the game with a set of states in a lookup table. Given a particular state of the board, the moves are considered the best responses for that situation. Opening books of some sort have been used in most Go programs.

A number of developments involving opening books in the 2000s involved using neural networks to create or test them. Huang, Conneil and McQuade [64] used neural networks in self-play to learn moves in the first 10 moves of the game through temporal difference learning and using high-level features. Lee [65] also used temporal difference learning in a neural network on a 19x19 board and Kendal et al. [66] look at using a neural network to learn opening game strategies on a 13 x 13 board.

An opening book for 9×9 Go was built by Audouard [67] using a grid coevolution technique. This utilized self-play of the MCTS player MoGo.

In 2010 Gaudel [68] discussed creating and analyzing opening books in Go while Mullins and Drake [69] implemented an opening book based on both fuseki and joseki with exact matching. Hoock et al. [70] created an ontology of good and bad openings by having professionals rate moves in the opening on a 5 point scale from very bad to very good on a 9×9 board. These are then matched up to provide an opening book based on fuzzy pattern matching.

An active opening book for 19×19 Go was created by Baier [71] in 2011 to assist MCTS . It uses an exact match of the board, or portion of the board to create suggested moves, but instead of choosing the move as the next action, it is used to bias the MCTS engine in-tree search. Bayesian learning was used by Michalowski [72] to improve the performance of the Fuego program by suggesting possible good moves in the first 20 plays of the game.

In 2014 a very large scale board position look-up was implemented using a Hadoop database system in [73]. Each move of every game was stored in all 8 rotations.

Steinmetz and Gini [74] describe a system of clustering together professional game positions to derive favorable moves in assistance of MCTS in-tree bias policies.

4.3 Monte-Carlo Tree Search

Since the early 2000s, much effort has been spent on exploring a technique that uses a stochastic model of a game tree to evaluate a static position on the board. Called Monte Carlo Tree Search, a version was first proposed by Brügmann [4] in 1993 . This technique completes games with random moves many thousands of time from a given move, and scores this move based on the outcome of this random sampling of games.

The mathematical foundation for the MCTS algorithm was introduced in 2002 by Auer, et al. [7]. It introduced the UCB1 formula in the context of the multi-armed bandit problem.

In 2003 Bruno Bouzy created two programs Olga and OLEG [75]. These used some Monte Carlo methods and simulated annealing. He also looked at various attempts at Monte Carlo approaches and modifications in [76] including an all-moves-as-first

approach along with progressive pruning. This work continued with an almost-flat Monte-Carlo approach in 2004 [77] which used MC evaluation to prune nodes and then move to another depth, prune nodes based on MC eval, and so on. Further refinements of this pruning approach, which resembles MCTS, was carried out in [78] which modified playouts using domain dependent knowledge, and in [79].

In 2006 a modification to the Monte Carlo algorithm for creating search trees was published by Kocsis and Szepesvári [8]. It was called Upper Confidence Bounds Applied to Trees (UCT), and chooses moves by iterating through the scoring of candidate moves with the normal Monte-Carlo algorithm but keeping track of the number of times successor nodes in the tree are encountered along with their scores. When a successor node has been visited enough times, it is promoted to the status of a candidate node itself, and all such nodes are scored not by the usual MC method, but by using the best score of their most promising "child" nodes.

Chaslot et al. [80] provide a MC search with move selection strategies that are not UCB1.

Coulom [81] introduced and named MCTS and the mathematics behind it in his Crazy Stone program. This was expanded by adding patterns in [82], where the programs Elo ratings were also discussed. In a little-used variant, Yoshimoto et al. [5] discusses a method of scoring a candidate in MCTS not just by the number of playouts won and lost, but also the amount (number of stones) by which these games were won or lost.

Also in 2006 Gelly and Wang [83] introduced their "Mogo" program and discussed basic MCTS with no additions. This was expanded to introduce first-play urgency in [84] along with using patterns to affect the playouts. Gelly and Silver [85] combined online knowledge in the form of RAVE values and offline knowledge in the form of priors (as biases to newly built nodes) in 2007 .

Drake [86] introduced his "Orego" program, adding a proximity heuristic and an opening book. [87] discusses four different in-tree heuristics and experimental results from their application. Chaslot et al. [88] brought expert knowledge in the form of patterns to the in-tree calculations by modifying RAVE values ahead of time depending on the pattern.

Cook [89] examined the speed of playouts on current hardware in 2007. His results

showed that a single processor, 2.8 GHz machine could execute one hundred thousand playouts in approximately three seconds.

In 2009 MCTS was used by Balla and Fern [90] to play tactical assault manuevers in a real time strategy game. Perez et al. [91] also applied MCTS to a real time strategy game, notably pursuing multiple objectives in that context.

Chen et al. [92] provided a survey of MCTS as applied to Go in 2009. Other survey papers have followed by Rimmel [93] in 2010 and Gelly et al. [94] in 2012. Browne et al. [95] released a comprehensive study of everything MCTS in 2012. This major work listed application domains, variants, and the history of MCTS in a well-referenced volume.

Drake [9] improved the performance of his Orego program by introducing the last-good-reply playout policy. This was improved by Baier and Drake [10] in 2010 by adding an element of forgetting recent responses which lost after they won.

Huang, Coulom, and Lin [96] improve playout parameters using simulation balancing while RAVE value are used to bias playouts (in addition to the tree policy) in "Biasing Monte-Carlo Simulations through RAVE values" [97] by Rimmel. Similarly, in 2013 Fernando and Müller [98] explore the differences of various playout policies used in Fuego while Powley et al. [99] also examine playout policies, especially using UCB1 during playouts while also using it as a tree policy.

In 2011 Gelly and Silver [100] give a thorough description of MCTS with RAVE. Baudis and Gailly [101] describe the Pachi open-source Go engine which notably uses RAVE and introduces dynamic komi.

Hashimoto describes an implementation of accelerated UCT, which gives more weight to games which were started from deeper nodes of the tree, in [102].

Bubeck and Cesa-Bianchi [103] along with Bubeck, Perchet, and Rigollet [104] analyze and refine the mathematics of regret in multi-armed bandits.

Niekerk and Kroon [105] look at using decision trees to generate features which can be used to influence MCTS while Ikeda and Viennot [106] explore the use of static knowledge to bias the tree search.

Graf, Schaefers and Platzner explore how MCTS reacts to semeai situations in [107] while the limits of MCTC in certain Go positions (semeai and seki) are explored by Huang and Müller [108].

Lorentz and Horey [109] apply MCTS to the simple game of Breakthrough.

In 2015 Graf and Platzner [110] explored adaptive playouts using reinforcement learning. Jin and Keutzner [111] used convolution networks and GPUs to improve the performance of an MCTS Go program.

The program AlphaGo, developed by the Alphabet-owned company Deep Mind [1], achieved a milestone by beating human champion players including the recognized world champion Lee Sedol in non-handicap Go in late 2015 and March of 2016.

4.4 Parallelization of MCTS

Cazenave and Jouandeau discuss three methods of parallelization in [112] and [113]. They use the term single-run parallelization to mean what we call root parallel, multiple-run parallelization to mean root parallel, but updating the information more often than just at the end, and at the leaves to mean leaf parallelization.

Chaslot, Winands, and van den Herik compare leaf, root, and tree parallelization in one of the first comparisons with “Parallel Monte-Carlo Tree Search” [114]. They propose tree parallelization and introduce both local mutexes and virtual loss as means of avoiding the bottleneck of locking. They also introduce the notion of “strength speedup”, which measures the strength of the improvement by comparing against the winning rate of a program given more time.

Kato [115] creates a system for leaf parallelization that is based on a client-server model, updating in the tree after the first result comes back (thus not waiting for the last one).

“The Parallelization of Monte-Carlo Planning” [116] by Gelly, Hoock, Rimmel, Teytaud, and Kalemkarian specifies the algorithms for MCTS, multi-threaded, shared memory MCTS (tree parallelization with mutexes), cluster MCTS with tree sharing, and finally cluster MCTS with only root sharing. They show some very limited results in terms of improving winning rates while trying each of these methods.

In 2009 Enzenberger and Müller [12] introduced a lock-free version of tree parallelization in Fuego.

In “A Parallel General Game Player” [117] Méhat and Cazenave explore four variations of root parallelization across a number of game domains including checkers and

Othello using a program named "Ary" which integrates a Prolog interpreter. They observed the best results in checkers along with more moderate improvements in Othello.

Rocki and Suda explored "Massively Parallel Monte Carlo Tree Search" [118] by presenting MPI methods to perform root parallelization over thousands of cores. They use MCTS over the game of Reversi, and implement root parallel on a very large cluster. Their results are measured in terms of number of playouts accomplished.

In Soejima, Kishimoto, Watanabe, "Evaluating Root Parallelization in Go" [119] the authors explore various aspects of root parallelization, running experiments for analysis of performance data. They compare the performance of root parallelization using two different methods of scoring: majority voting and average selection. They use a modified version of Fuego playing on both 9x9 and 19x19 sizes. The results show winning percentages of average selection and majority voting against both sequential Fuego and sequential MoGo. Further results compared the lock-free tree parallelization directly against 64 node majority-vote root parallelization using Fuego. Finally, All these methods were engaged in picking moves in particular game situations against an oracle selection made by a very long time run of Fuego (80 s). They concluded that majority voting outperformed average selection in root parallelization, but that lock-free tree parallelization outperformed root, and that root did not scale well past 32 cpus.

In "Scalability and Parallelization of Monte-Carlo Tree Search" [120] Bourki, et. al. explore the limitations of parallelization with multi machine (messaging) implementations on clusters. They discuss slow parallelization versus very slow root parallelization conclude that the slow version has an advantage over very slow. Additionally, they discuss some of the weaknesses, in Go, of MCTS players, namely semeais.

Fern and Lewis [121] looked at a different formulation of root parallelization which they called Ensemble Plannning over five simpler game domains.

In "Scalable Distributed Monte-Carlo Tree Search" [122] Yoshizoe et al. used a depth-first UCT and transposition-table driven scheduling in order to create a distributed version of Fuego which ran on up to 1,200 cores.

Niekerk and Kroon [123] implemented both tree and root parallelization on their Go player, Oakfoam, and looked at the results in terms of number of playouts per second.

Nishino and Nishino [124] implement leaf and "sample" parallelization on an imperfect information game Daihinmin (a card game).

Goli et al. [125] look at using MCTS to statically map out parallel tasks onto both CPUs and GPUs.

In [126] the use of teams of software agents to pick moves is explored, similar to root parallelization with majority voting. This is also examined by Marcolino and Matsubara [127] and then expanded by Marcolino et al. [128].

In "Parallelization of Information Set Monte Carlo Tree Search" [129] Sephton et. al. compare root, tree, and leaf parallelization in a card game "Lords of War" which is an imperfect information game with a branching factor from 25 to 50 and a depth of about 60 turns. They utilize Information Set MCTS which tracks a game state including hidden information not as a single random pick of what the hidden information could be (my opponent is holding three kings) but as the set of all possible versions of the hidden information. At each simulation the determination of the hidden information is made, rather than just once. This simulates a search across a wide range of possible combinations of hidden information. The testing done compares the efficiency as measured by the speed of the algorithms to produce a tree of 5000 nodes. This is done with both MCTS and ISMCTS across root, tree, tree with virtual loss and leaf parallelization. They show little difference between MCTS and ISMCTS and show that root parallelization appears to be the most efficient.

Schaefers and Platzner [130] in "Distributed Monte Carlo Tree Search" analyze the effects of using a parallel transposition table for tree parallelization along with dedicating some compute nodes to broadcast operations to help scaling to large numbers of nodes. They explain their parallel architecture with its information flow and the distinct jobs of different parts of the hardware in great detail.

In 2015 Gopayegani, Dusaric, and Clarke [131] apply parallel MCTS to the management of energy demand and production.

In "Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors", Mirsoleimani et. al. [132] compare scalability of the root and tree algorithms on modern Intel processors over the game of Hex. They show that mutex locks are not limiting even up to 16 threads if implemented on the Intel Xeon Phi hardware which contains 30$\times$ faster communications than a typical Xeon processor.

4.5 Computer Hex

Originating in the world of mathematics, the game of Hex has been played by computers from early its development, including even an analog computer that used the measured the current differences of circuits through the board in order to create a heat map and thus choose the best position for the next move [133].

In 2000 Anshelevich [134] used a theorem-proving approach to discover virtual connections, and combined this with an $\alpha - \beta$ search algorithm to create the "Hexy" program. This program was later modified with a set of hierarchical deduction rules in [135]. The resulting version was a world-champion computer Hex player, and played at a reasonably strong rating in an online forum.

In 2002 van Rijswijck [136] used pattern search and graph distance heuristics to build a computer Hex player called Queenbee.

Yang [137] built a partial solution to 7×7 games of Hex, giving some winning and losing first moves. This size board was later fully solved by Hayward et al. [138] using heuristics to reduce the state space by filling in part of the board.

The programs "Six" [139] in 2003 and later "Wolve" [140] in 2008, both $\alpha - \beta$ based algorithms with sophisticated evaluation functions continued improving the quality of computer-Hex programs.

Rasmussen developed a sub-game decomposition algorithm for playing hex which he dubbed H-Search [141]. Other algorithmic improvements include refinements by Henderson and Hayward that enabled the pruning of a class of moves through probing along the opponents in a so-called 4-3-2 position [142], and the introduction of the notion of decisive moves and anti-decisive moves for MCTS in the context of Hex by Teytaud and Teytaud [143].

A survey of the mathematical background to the game and the combinatorics involved for computation of solutions was published by Hayward [144] in 2006.

The MCTS Hex player "MoHex" was created by Arneson et al. [145] in 2010 . It utilized reduced equivalent boards in the playout phase rather than the actual game board, and pruned provably inferior children in the game tree. MoHex was improved in 2013 [146] by the addition of learned patterns to improve both the in-tree prior values (bias values) and the out of tree simulation policy.

Chapter 5

Smart Start

In this chapter we will show our work developing an improvement to the way the opening moves in a game of Go may be played by a computer.

When humans play traditional, well-researched games such as chess, checkers, and Go they often rely on remembering a set of fixed sequences of best moves called an opening book. A book will usually list one or more options for a player to use when the situation on the board has exactly a particular configuration. By following the prescriptions in an opening book, players can quickly make the highest quality moves available without doing much analysis.

5.1 Proposed Approach

We have created the SmartStart method to act as a generalized opening book, providing higher quality moves in the initial play of the game without requiring an exact full or partial board match to known games.

We first record the moves chosen during games between professional players for each board position of the initial moves of the games. We then group these records, within each move number, into a small number of clusters based on the similarity of the board positions. Each cluster so created contains a list of all the next moves chosen in the records of that cluster. During play, we determine which cluster is closest to the current board situation and utilize the next moves in that cluster to guide the Monte Carlo style algorithm.

5.1.1 Opening books in Computer Go

One of the ways in which computer chess programs have succeeded in reaching the grand master level of the game is through a basic method of learning from human experience for the opening moves of a game. Long before computer chess programs entered the scene, most serious students of chess would spend hours studying the opening moves of master level players. Because the grand masters had discovered and developed over the years the most powerful sequences of opening moves, one of the fastest ways to improve one's game was to study these opening books and memorize exactly the "best" move given the current position. Computer chess programs, like well-studied humans, can therefore be loaded with these opening book libraries, and may simply do a look-up of the best move given a particular position. With the ability to do a table-lookup of the "correct" move at the beginning of the game, chess programs have been spared the arduous task of calculating their own moves at the beginning of the game when a game tree may need to be longer and wider to achieve the same kind of powerful result that a smaller tree might achieve later in the game.

In the game of Go, although there are well-known patterns of play in the opening moves and numerous books regarding opening strategy [147], lengthy and comprehensive opening books comparable to those in chess have not arisen, due to the large number of possible top-quality opening move sequences. If one limits the scope of the moves to one corner of the board, however, a set of literature exists explaining move sequences called joseki ("fixed stones" in Japanese). These are mini opening books concerning only a small area of the board, typically sequences of moves in a corner.

5.1.2 Problems with Opening Books

The way an opening book works in the code of a computer Go program is that sequences of moves are placed into a database so that they can be looked up by the board position that would exist after each of the moves in the sequence. So for a sequence of six moves, A B C D E F, the algorithm would find this sequence if it was looking at a board with just move A on it, or a board after moves A and B, or the board as it would look after moves A, B, and C, and so on. If the lookup succeeds, the next move in the sequence is then the move of choice from the opening book. For example, if it is move 5 with black

to play (since black always moves first in Go) and the board matches the board state that would exist after the moves A B C D, then the move E is the one which will be chosen by the opening book. For the first move of the game onto an empty board, one of the sequences (often the longest) is designated as the choice to be made.

For any given board state, there is at most one entry in the opening book. However, there are many variant sequences that begin similarly, as opponent play cannot be controlled. The sequence A B C D E F may be accompanied by another sequence that begins the same but then branches such as A B C G H, or A I J K L M.

Once a play has been made outside of any of the sequences contained in the opening book it is no longer possible to find a move which will match, and so the opening book will no longer be consulted.

Opening books exist in many computer Go programs, including some with very long sequences of 30 moves. If two programs play each other using the same opening book, the first moves of the game are then completely deterministic, consisting of exactly the moves in the sequence designated to make the first, empty-board, move. When a program with an opening book plays against one with no opening book, however, the actual number of moves used from the book usually ends up being quite small, and play leaves the book quickly. This can mean that the presence of a traditional opening book has very little effect on play against a non-opening book opponent.

In testing we found that black with an opening book playing against an opponent with no opening book used only the first move in 84% of the games and used only the first and third moves in 12% of the games. Playing white with an opening book against an opponent with no opening book resulted in using no moves from the book 6% of the time and only one move from the book 76% of the time. To see how this affected the overall quality of play, we tested a version of Fuego in two tournaments, once with an opening book, and once without, against a version of Pachi with no opening book. We found that the opening book used in this fashion was not able to provide a statistically significant improvement to the resultant winning rates (see Table 5.1). With a null hypothesis that the opening book in Fuego does not change Fuego's ability to win, we found a two-tailed p-value of 0.86, which does not allow us to reject the null hypothesis (usually rejected with a a p-value of 0.05 or smaller).

Table 5.1: Fuego vs. Pachi 10. 10,000 games per tournament.

	As Black	As White	Cumulative	p-value
Fuego with no Opening Book	36.3%	41.1%	38.7%	-
Fuego with Opening Book	36.0%	41.6%	38.8%	0.862

5.1.3 Using Professional Play in the Opening Game to Improve MCTS

In this chapter we show how the play of MCTS-based computer players can be improved in the opening moves of Go by SmartStart, our technique to utilize a full board pattern match against clusters of positions derived from a large database of games played by top-ranked professional players. While an opening book requires a perfect match of the current board in play against a position found in a database, SmartStart matches the current board with the nearest cluster of positions in a database. By matching against clusters of similar moves instead of seeking a perfect match, our method is guaranteed to find some match, and the solutions found are more general.

Instead of choosing a single move on a board with a perfect match to an opening book position, SmartStart constrains the Monte Carlo search tree during the initial moves of the game to only those moves played by professionals in games most closely matching the position of the game in question.

During a Monte Carlo search, all legal moves are considered at each turn during a playout. This means that at the beginning of the game, the search will consider all 361 initial locations on the board for move 1 (ignoring symmetry), then 360 for move 2, and so on. The engine must then play out a number of games to the end of the game for each of these possibilities in order to evaluate them.

Because so many different moves are being considered that are not at all viable, the MCTS engine is spending a significant amount of effort on needless playouts, and occasionally picks moves which are easily recognized as ineffective even by novice players.

Our work involves limiting the search by considering only those moves that have been made by professional players in the same or similar situations. This approach is different from using an opening book in two ways: we are not using an exact match of the board, and we are not directly picking the move with a match, only reducing the search space of the Monte Carlo engine and letting it conduct its playouts to determine

which next move has the best chance of winning.

The scale of this reduction at the beginning of the game can be substantial. For example, if we consider a game after eight moves a normal search for the next black move would start sub-trees at all open intersections on the board, $(19 \times 19 - 8)$, which is 353 different possibilities. When using SmartStart, with 64 clusters for example, the average number of next moves that would need to be considered in this situation after move 8 is about 34. By only considering these moves, we reduce the number of next move possibilities by a factor of 10.

5.2 Design of SmartStart

The overall design of SmartStart involves matching up the current state in a game being played to a cluster of archived games that are "close" or identical to the game being played. Within the matching cluster of games, which were drawn from professional play, there will be a grouping of next moves which SmartStart uses to assist in the decision for the move choice in the current game. The creation of the clusters and storing the relevant information about those clusters is done offline, and so takes no time during game play. The lookup during the game to find the closest cluster center to a board state involves comparing the board state with the board state associated with the center of a given cluster and finding a distance between the two states. This must be done for each cluster available, and once the closest cluster is found, the data about its next moves are used to calculate the next move for the current game.

5.2.1 Game State Representation

To compare various board states, we must have a mathematical representation of the state, along with methods to measure the difference or similarity of a pair of states. In SmartStart we use a representation where we consider each of the 361 points on the board to be two separate elements, giving each board state a representation as a 722 $(2 \times 19 \times 19)$ element vector. Each element of the vector represents either the presence or absence of one color of stone at one of the intersections on the board. Thus the first element of the vector would represent a black stone in the lower left corner of the board, while the second element would represent a white stone in that same location, as shown

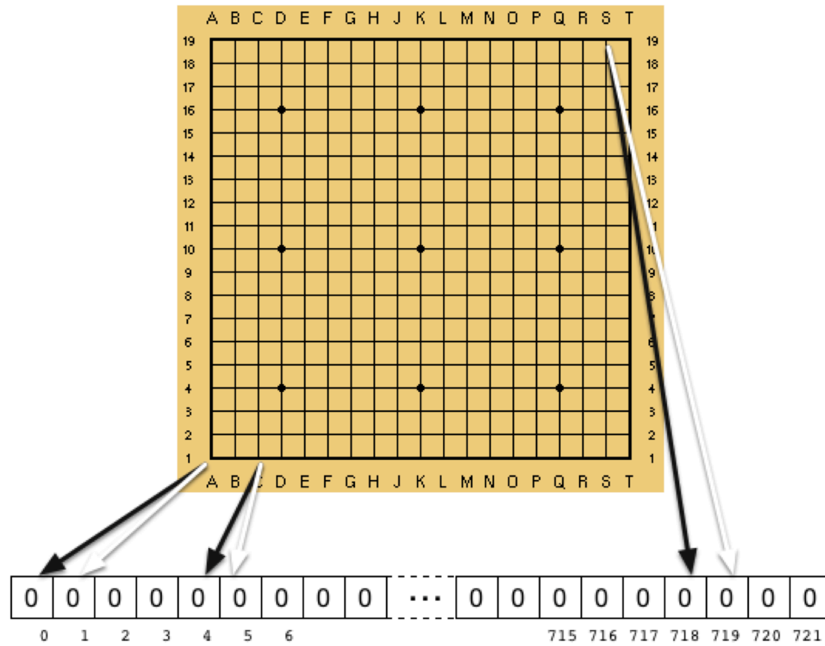


Figure 5.1: Board positions in the vector representation.

in figure 5.1. By using two elements for each location, one for black and one for white, board situations where black and white stones are swapped do not appear to be similar, or too dissimilar, as they do when a single location is considered a single element in the vector.

Because of the geometric symmetry inherent in the game of Go, we also need to consider the equivalence of symmetric board positions. Because a board state is defined as the locations of stones on a square grid, there is an 8-way symmetry: four rotations of 90 degrees and another four rotations of a mirrored position, as shown in Figure 5.2. Although the locations of the stones is different in each of these 8 boards, they nonetheless all represent an equivalent game. In a comparison of two board states, a source and a target, either the source or the target must be rotated through all eight symmetric orientations to check for an equivalence match.

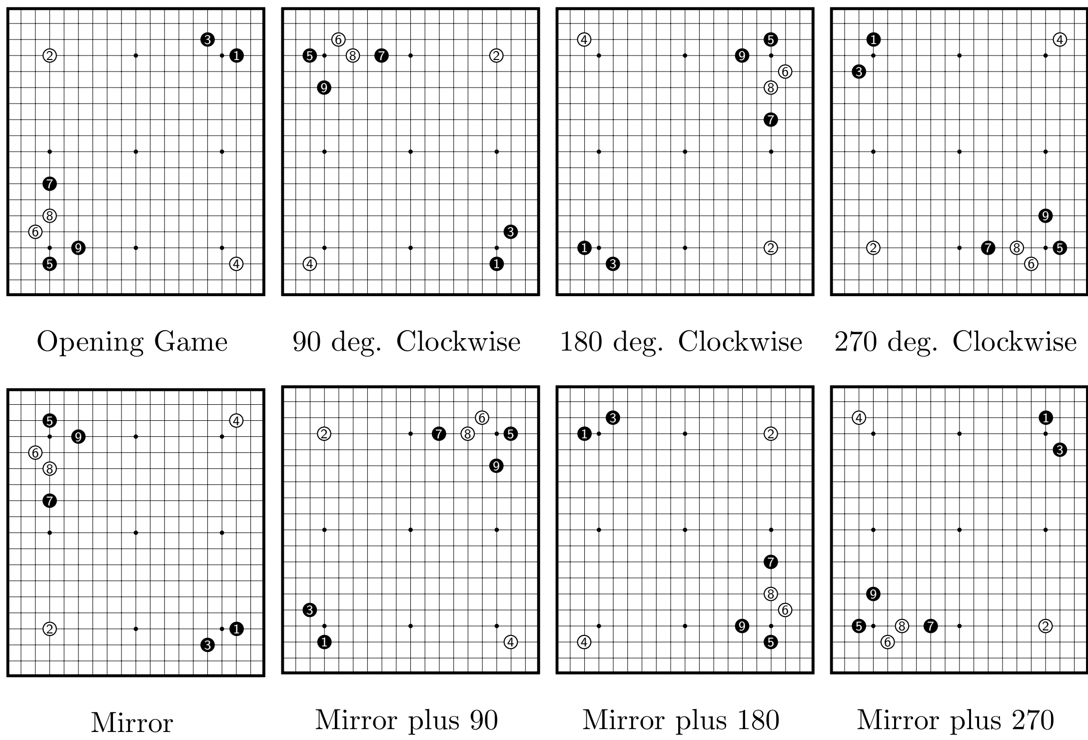


Figure 5.2: A board position in the 8 symmetric orientations.

5.2.2 Database Creation

In order to find similar game positions in games played by professional players, we have scored professional games from a large database and then clustered these together based on their similarity.

A sample of games spanning from 1980 to 2007 from a large database of full 19×19 board professional level Go games was used as the basis for finding patterns in opening play. We created results from a 1046 game sample of the commercially available Games of Go on Disk collection. For each professional game in the sample, and for each of the first twenty moves in those games , a vector representation of the board was created as described above. Along with this board representation, the professional's response to that game state was also recorded. Each of these representations along with the next move played was then reproduced in all eight rotations so that each professional game created 8 entries for move 1, another 8 entries for move 2, another 8 for move 3 and so on for the first 20 moves of the game, and a total of 160 entries per sample game. We increased the number of entries in the database eightfold rather than rotate a current move through the eight symmetries at runtime in order to avoid the rotation cost during the runtime phase. The same number of comparisons will take place at runtime with either architecture.

Each rotation of each move of each professional game was entered in the database as a 722 element vector of ones or zeroes as described above. In order to best facilitate quick and accurate matching, the database has been divided into separate databases for each move number. Because move number one is always black, and move number two is always white and so on, each of the databases will have groupings most appropriate to the given move. That is, if we are looking for the best move at move number 8, there should already be seven stones played on the board (barring some very early capture). The database of board positions that will be searched against in this case is precisely the set of board positions after move number 7. The advantage of a divided database is that we can be assured that only the most similar board positions are being searched.

Finally, within each move number, we took all the entries for that move (8368 for the 1046 game sample) and clustered them into a much smaller (64 to 256) number of clusters.

5.2.3 Clustering of Database Games

One of the important ideas of SmartStart is that instead of searching through a history of professional play for an exact match of the current game state in order to find a good next move, a search will be conducted over a smaller number of groups of game states (sampled from professional play), where each group consists of a set of games which are similar to each other. These should be more similar to each of the other games in their own group than they are to members of one of the other groups. Each of these groups can then be expressed as the position average of the members of the group. In the case where these games are vectors of length 722 then, this average represents the center of the groups' games in 722 dimensional space. By searching only through each of the groups' center positions, we find a group of games that are similar to the current position, and do so with many fewer comparisons than needed if we look through all the games individually. Additionally, this method is guaranteed to find some matching group, the closest one, whereas attempting to find an exact match can fail if such an exact match does not exist in the database.

To accomplish this, the entries from the professional games for each move number were clustered into a small number of groups based on the similarity of the entries as measured by the cosine of the normalized position vectors. If we had entries with only two elements each, this would equate to finding the cosine between the angles of these entries in an x-y space. Because of the normalization, the entries would all be on the unit circle. The cosine value of a perfect match of two vectors will be 1.0. The number of groups (clusters) is chosen before the clustering, and we experimented with using 64, 96, 128, 160, 192, 224, and 256 clusters. Note that because of the eight-fold symmetry the number of clusters is eight times the number of different scenarios.

Using small tournaments of a program modified with SmartStart we found only a small variation in the winning ratio when changing the number of clusters, and so used the 64 cluster and 96 cluster databases in this work. Both the number of entries in the database and the number of clusters affect the number of next moves (the move suggestions) supplied by each of the clusters. As the number of clusters decreases, it is more likely that in each cluster there are many different next moves. This makes the prediction of the next move more difficult.

The clustering was accomplished with the Cluto program [148]. This program provides two different ways of creating clusters. A top-down partitioning method begins with all entries in a single cluster and then goes through multiple rounds of bisecting the current clusters until the desired number of clusters is reached. A bottom-up agglomerative method begins with all entries in their own clusters and then repeatedly merging nearby clusters until the desired number of clusters is reached. When the number of items to be clustered and the number of clusters are both large, these two methods end with very similar results, with a slight edge for partitioning [149]. Therefore we chose the top-down bisection method. For the similarity measure we used the commonly adopted cosine measure.

The bisection method of clustering works as follows: all entries are first considered to be in a single group. This group is then divided into two clusters by movement of each entry into one of the two new clusters. The choice of which cluster to place each entry in is determined by maximizing the similarity in each cluster, where the similarity of the cluster is based on the summation of the similarity of each pair of entries. Specifically, at each bisection this algorithm tries to maximize

$$\sum_{i=1}^{k} \sqrt{\sum_{v,u \in S_i} sim(v,u)} \tag{5.1}$$

where k is the number of clusters, v and u are entries, S_i is the ith cluster, and $sim(v, u)$ is the similarity measure of the two entries v and u. In this case our similarity measure is the cosine. The term inside the root here is the sum of all similarity measures inside a single cluster, and the algorithm maximizes this over all clusters. After the fist bisection, the process continues by bisecting each of the resultant clusters until the desired number of clusters is reached.

For each of the final set of clusters created by Cluto, we created a single cluster entry in our comparison database to be used during play. Each of these entries contained the following information: the normalized vector representing the cluster's center, the total number of entries that are contained in the cluster, a list of the next moves played during the games that are included in that cluster, and the frequencies of those next moves.

We investigated the accuracy of these clusters by considering how often they contain a next move which predicts the next move found in other games taken from the GoGoD

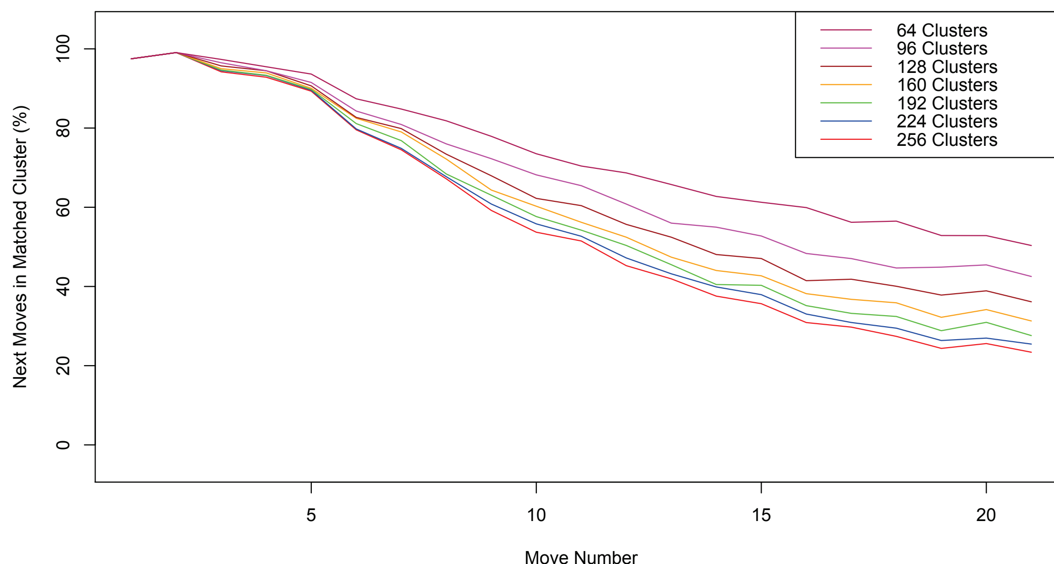


Figure 5.3: Percentage of next moves found in closest cluster.

database. We selected a random sample of approximately 8000 games which were not used in constructing our clusters, and then at each position in those games found the closest cluster as explained in the next section. We consider a success to have occurred when the cluster contains, as one of its next moves, the move actually played in the observed sample professional game.

We found that the success rate decreased as the move number increased. We also found that the success rate decreased with an increasing number of clusters. These results are shown in Figure 5.3. Since the average number of games per cluster also decreased with increasing number of clusters, it is not surprising that our success rate would also decrease.

While it is an advantage to have a smaller number of next moves in each cluster so that our filtering or biasing has a greater influence, it can be a disadvantage because it becomes somewhat less likely that your cluster will contain the "correct" move. Since even with a larger number of next moves we are filtering out or biasing against a very large number of unacceptable moves, we elected in our experiments to first look at using a smaller number of clusters and their larger sets of next moves. As mentioned above, we found with small samples very little difference between using fewer or larger numbers

of clusters.

5.2.4 Finding a move during play

At each point during a search where the MCTS algorithm was searching through the initial moves of the game, both for candidate moves, and during playouts to evaluate the candidate moves, we limited the options of the program to those moves that had been played by professional players in the games contained in the nearest cluster to the current position. The position of the game board being evaluated in the search is expressed as a 722 element vector of ones and zeroes identical to the scoring of the professional games before creating the clusters. We then compare this vector to each of the vectors representing a cluster center in the comparison database. In order to compare the two scores, current position and cluster center, the cosine of these two vectors is calculated. Whichever cluster has a cosine closest to the value one is the nearest. The cluster whose center was nearest to the given position was then selected to provide the candidate next moves. These candidate next moves from the games contained in the nearest cluster were then used to modify the MCTS process.

We looked at two ways to modify the MCTS search: filtering and biasing. In filtering the choice both in and out of the search tree is limited so that only the moves from the cluster's list are allowed to be chosen. Thus instead of looking at all the possible moves (over 300 at the beginning of the game) the algorithm considers the few (usually ten to forty) allowed by the filter. In the biasing method we change the in-tree node selection by modifying the statistics of the preferred moves. Used in [85] and often called "priors" for prior knowledge, this technique seeds the selection statistics with virtual win and visit counts. As nodes in the tree are created, instead of beginning with 0 wins from 0 visits, they are biased by beginning with, for example, 40 wins from 40 visits. Using this method means that although all possible moves are open for consideration by the MCTS algorithm, the moves selected by SmartStart are given a strong preference since they will appear to have already won a large number of playouts.

The positions that were considered were those up to move twelve. We also explored using the knowledge up to move twenty, but found that the overall win rates actually declined. This corresponded with a similar decrease in the quality of the clusters that had been produced: after move twelve the tightness of the clusters and the clarity with

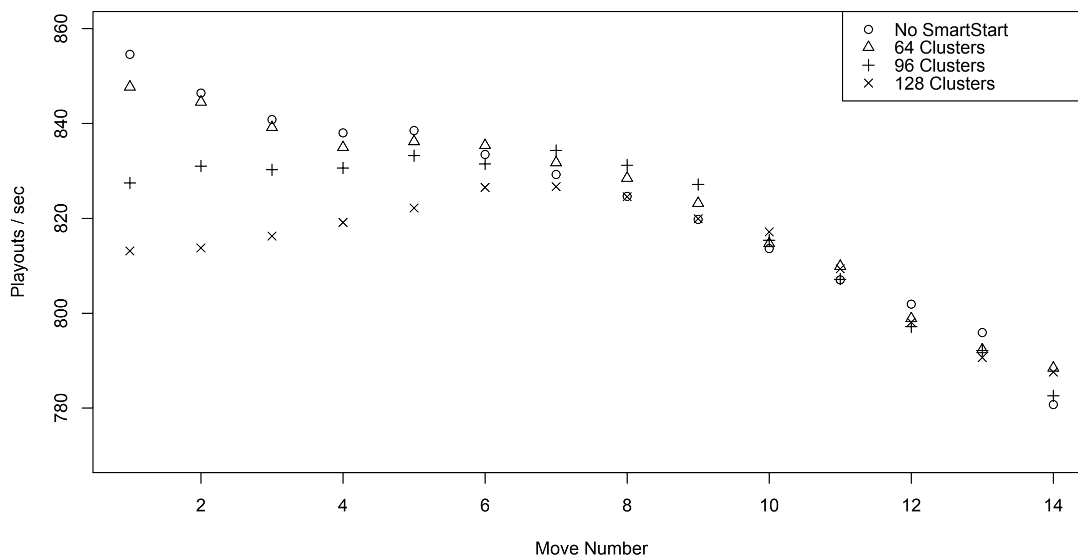


Figure 5.4: SMARTSTART Speed in playouts/s.

which they can be distinguished from other clusters gradually decreased.

The computational overhead of creating a position's score and finding the closest cluster for the initial moves was small but increased with the number of clusters. As measured by playouts per second, comparing to 64 clusters was less than 2% slower while 128 clusters was less than 5% slower at their worst, which was the first few moves when the most comparisons would take place. The speeds are shown in Figure 5.4.

5.3 Current Open Source Go Engines

There are currently three popular and strong open-source Go engines available which use Monte Carlo techniques, Fuego, Orego, and Pachi. Additionally, the open-source Gnugo program, though weaker than these, is often used as a traditional AI techniques opponent. Fuego is built using C++, Pachi with C, and Orego with Java, while Gnugo is written using C. Of these, Fuego is considered the strongest.

The algorithms for Monte Carlo Tree Search can be categorized along a few different axes: the random play out policy, the tree selection policy, and the propagation method of results.

The policy during random playout is the amount of control exerted over the choice of moves in the random playout phase. This is known as the default policy. When the engine picks only totally random legal moves, this policy is considered to be a "light" policy. When a heuristic or set of heuristics is applied to modify the random move selection, the policy is considered to become "heavy": the more heuristics applied and the greater the filtering effect, the heavier the policy. Heuristics range from simply not filling in eye spaces, to relying on extensive pattern matching to filter available moves.

Another distinguishing feature of a Go engine is the techniques used to choose which nodes on the edge of the search tree to expand. This is known as the tree policy, and determines the fashion in which the tree is built. The nature of the values used to choose from which node to start a simulation, and how to calculate and modify them, are often the heart of the various named versions of MCTS such as flat UCT and All Moves As First with its most popular version RAVE.

Finally the method in which a win or a loss from a playout is propagated through the tree differentiates the different algorithms used. Whereas in UCT the result moves up the tree from the node at which the default playout began up to the root through each nodes' parents, in All Moves As First algorithms, the result is also added to statistics in any node in the tree that could have been played (but was not) that matches some move occurring during the random playout phase.

5.3.1 Fuego

Fuego [150] is a Go Engine and game framework written in C++ created and maintained by the Computer Go group at the University of Alberta, Canada. It consists of a number of modules to run a go board, read and write SGF game files, run a game-independent MCTS game, and run an MCTS go game engine using various heuristics including RAVE. Fuego contains the ability to invoke an opening book if desired, and that book can be replaced or modified. Additionally, Fuego has the ability to be run as a multi-threaded application. Fuego has done well in computer to computer Go tournaments, with an occasional top place, and often scoring in the top three.

5.3.2 Orego

Orego [151] is a Go Engine written in Java created and maintained by Peter Drake and his students at Lewis and Clark College in Oregon. Orego is an object-oriented implementation which uses a class hierarchy to create different kinds of MCTS players.

The classes which implement players determine the kind of tree search which will take place. There are classes for the traditional flat Monte Carlo search, an unadulterated MCTS tree, a RAVE equipped search tree and one using last good reply techniques.

Other classes directly implement the playout policy, with three heuristics available: escape, pattern, and capture. The playout heuristics can be specified at runtime with any priority order and any probability of being invoked on a particular move pick.

5.3.3 Pachi

Pachi [101] is a Go Engine written in C mostly by Petr Baudis as his Master's Thesis work at Charles University in Prague. It is maintained and upgraded as an open source project. Though written in C, Pachi is divided into a number of modules which together implement a Go specific MCTS engine.

Pachi uses simplified RAVE statistics in the tree policy.

Pachi implements a semi-heavy playout policy, using a small (less than 10) set of heuristics in a given priority order to generate lists of candidate moves. Each heuristic is given a 90% chance of being consulted. If there are any moves in a list produced by a heuristic, one is chosen from that list.

5.3.4 Gnugo

Gnugo [19] is a Go Engine using traditional, not Monte Carlo, techniques. It has been in intermittent development since 1989, with the most work ceasing after 2009 as MCTS programs became significantly stronger.

Similar to many traditional Go programs, Gnugo uses a number of independent analysis modules to study a given board situation, and then generate a list of reasonable moves. Then using estimated territory values for those moves, along with some heuristics dealing with strategic effects of the moves, it picks one of the candidates as the next move. It does not conduct a global lookahead search.

5.4 Results

We tested our method by incorporating it into a pair of strong open-source engines and compared win rates of modified and unmodified versions of these programs versus other Go-playing programs. For exploring the different implementation parameters mentioned above and for part of the following results we used the Orego [151] program, written in Java. We played Orego against the older, non Monte Carlo Go program Gnugo [19]. We also incorporated our method into the stronger Fuego [150] program, written in C++, and played it against another strong open-source MCTS Go program called Pachi [101].

5.4.1 Statistics

One of the problems with trying to study the end results of Go is the large number of games that must be played in order to detect a difference in playing strength. In any situation where the results are binary such as win-loss records of games, the formula for the 95% normal approximation of the binomial confidence interval is $p \pm 1.96\sqrt{p(1-p)/n}$ where p is the probability of a win and n is the number of trials in the sample. This means that 95% of the time an experiment is run the actual result will be within the confidence interval of the value seen in the sample seen by the experiment. Given this formula a very large number of games must be played to determine a reasonably precise value of the software's playing ability . For example, to get an approximately 2% confidence interval when the winning rate is about 50% requires 10,000 games. In our experiments we have used 14,000 game tournaments in order to provide a 95% confidence interval of approximately 1.65% ($p \pm 0.825\%$).

When comparing two different versions of a program we are usually concerned with determining whether one version is performing better than the other. The simplest way to do this would be to observe a large enough difference such that the 95% confidence intervals of each of the two programs do not overlap. This is actually a too stringent measure when comparing two win ratios. A much more accurate measure is to use what is called the two-proportion z-test, or a score test of binomial proportions. In the case of testing computer Go programs, we wish to determine if one proportion of wins is greater than another in a statistically significant way. This test tells us the "p-value": the probability that the result we see in testing is due to normal variation (called the

null hypothesis: the programs are really of equal strength and we are seeing a normal variation of this), or if the result we see in testing is so unlikely that we should conclude the opposite (called the alternative hypothesis: that one program is stronger than the other). By convention when the p-value is 5 percent or less we reject the null hypothesis and call the probability of the alternative hypothesis “significant”.

For example, to see if Fuego can win against Pachi more often than Orego wins against Pachi, we state our null hypothesis (the theory that we are trying to disprove) as WinRate(Fuego) $\leq$ WinRate(Orego) and so our alternate hypothesis is then WinRate(Fuego) $>$ WinRate(Orego). If Fuego won 510 out of 1000 games while Orego won 500 of 1000, we would find a p-value of 0.3437. This means that we have a 34.37% chance of finding these experimental results given our null hypothesis that Fuego was equal to or weaker than Orego. By convention, therefore, we do not reject the null hypothesis. If Fuego won 5100 out of 10,000 games and Orego won 5000 out of 10,000, though, we would find a p-value of 0.08074. That would mean we still have a 8.074% chance of finding the result we did given our null hypothesis, and so we still accept the null hypothesis. Finally at 14,000 games, if Fuego won 7140 while Orego won 7000, our p-value would be 0.04831 which means we now have only a 4.831% chance of finding this result under the null hypothesis. Thus we reject that and accept our alternative hypothesis that Fuego is stronger than Orego.

Figure 5.5 shows the various p-values for a 51% winning rate versus a 50% winning rate, based on the number of games played. The p-value shows the percent chance that the player winning 51% of its games is equal to or weaker than the player winning 50% of its games.

5.4.2 Experimental Parameters

We first incorporated our SmartStart technique into the Monte Carlo style “Orego” program. The two versions of Orego, with and without SmartStart, were matched up against Gnugo 3.8 at skill level 10. The games were played running both the Orego and Gnugo programs on the same machine, moderated by the “gogui-twogtp” script [152]. This script communicates with each program using Go Text Protocol (gtp) and accumulates the results of all the games played. Both versions of Orego played with the number of playouts per turn fixed at 8,000. The version of Orego with SmartStart

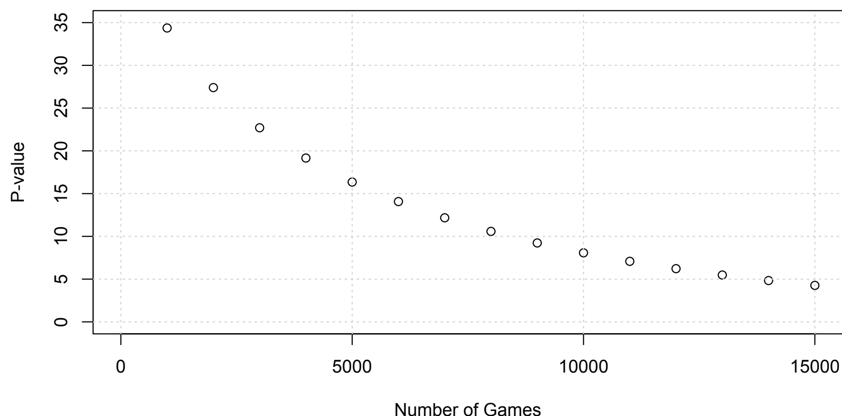


Figure 5.5: The p-values calculated for comparing a 51% winning rate versus a 50% winning rate.

was played with SmartStart invoked through move twelve using a database containing 96 clusters.

Since the Fuego program is a stronger computer Go engine, we also ran both the normal Fuego and a SmartStart Fuego against an opponent, in this case the open source program Pachi v.10. Fuego played with the number of playouts set to 16,000. These tournaments utilized a database divided into 64 clusters for each of the games' first twelve moves.

Our Fuego testing was divided into two different tournaments. In the first tournament the professional responses were used, as they were in the Orego tournaments, to filter the moves available to the Fuego engine, both during the in-tree decision process, and during the playouts.

In the second tournament, the matching responses were used to bias in-tree nodes. From a given position represented by a tree node, the moves which had been chosen by professionals were given a bias of 40 victories out of 40 games played. This bias causes the Monte Carlo algorithm to favor exploring those nodes over those without the bias.

All the games in all the tournaments were run using Chinese scoring rules and a 7.5

point komi on a 19×19 board.

5.4.3 Orego vs. Gnugo Results

In our tournaments of Orego vs. Gnugo 3.8, we played 14,000 games of unmodified Orego against Gnugo, and 14,000 games of SmartStart Orego against Gnugo. In both of these tournaments, half the games were played with Orego as black, and half the games with Orego as white.

The win rates of Orego and Orego with SmartStart are shown in Table 5.2. With the 14,000 game tournaments and the given winning rates, we have a one-sided p-value of 0.0119. This means that there is only a 1.19% chance that the SmartStart Orego winning rate is equal to or less than the unmodified Orego winning rate. Thus we have a statistically significant improvement in the winning rate achieved by applying the SmartStart method to Orego.

Table 5.2: Win Rates of Orego vs Gnugo. SmartStart Filtering applied through move 12.

	As Black	As White	Cumulative	p-value
Unmodified Orego	40.76%	40.06%	40.41%	-
SmartStart Orego	43.04%	40.44%	41.74%	0.0119

5.4.4 Fuego vs. Pachi Results

In the Fuego vs. Pachi tournaments, we started by matching up Fuego with our SmartStart algorithm configured for filtering. 14,000 games were played with this configuration, and 12,000 games were played with an unmodified version of Fuego vs. Pachi. 14,000 games each were played using the Fuego engine modified with our SmartStart algorithm providing a bias of 40 wins instead of filtering.

The unmodified and SmartStart versions of Fuego were all limited to 16,000 playouts per move. The unmodified version of Fuego was run with its opening book disabled.

The win rates are shown in Table 5.3 along with the p-value for the null hypothesis that the SmartStart Fuego version is equal or weaker than the unmodified Fuego.

Thus in applying SmartStart both as a filter and as a bias we obtained a small but statistically significant increase in winning rates.

Table 5.3: Win Rates of Fuego vs. Pachi 10. SmartStart Filtering and Bias applied through move 12.

	As Black	As White	Cumulative	p-value
Fuego with no Opening Book	50.88%	57.66%	54.275%	-
Fuego with SmartStart Filtering	54.16%	56.54%	55.35%	0.0424
Fuego with SmartStart Bias 40	53.97%	57.71%	55.84%	0.0058

5.5 SmartStart Summary and Future Work

By applying the SmartStart technique to two different open-source programs, Orego and Fuego, we were able to create statistically significant improvements in the winning rates of both of these programs over their opponents. This was true for using the SmartStart knowledge either as a filter or as a large bias.

Continuation of this work will include testing the effects of SmartStart when the number of playouts per move is larger through the entire game, and applying the system to the Pachi go engine. Additional testing of different numbers of clusters will be conducted to more thoroughly examine the effects of large versus small numbers of clusters, including increasing the number of clusters with the move number to represent the greater possibilities available at each move. Finally we will also explore the effects of using larger samples of expert games to create the clusters.

Chapter 6

Parallel MCTS

Monte Carlo Tree Search has proven very successful in a number of domains, and due to the nature of running many independent playouts it is an algorithm that lends itself well to parallelization. As described above in chapter 3 there are generally three ways to parallelize search in an MCTS tree: at the leaves of the search tree, throughout the entire tree, and at its root.

Over the last ten years, commercial computer processors have not become much faster in terms of their clock speed, but more numerous; dual core processors replacing single core, and being replaced by quad core processors in turn in most home and business configurations, and 8, 12, 16 and more core processors in many kinds of server chips. This trend means that in order to improve performance of MCTS in the future, especially in applications where time is of the essence and one cannot simply wait longer for a better answer, parallelization will become the key ingredient.

In this chapter we will explain our work on a detailed, in-depth comparison of the efficacy of the three kinds of parallelization. Additionally we will introduce a new variant of root parallelization which uses the idea of heterogeneous algorithms to create a diversity of search trees from which to derive a solution.

6.1 Comparisons of Parallelization of MCTS

Some direct comparisons have been run over the years over differing methods of parallelization. These studies have almost all been conducted in the domain of Go, since

most MCTS implementations were first done in this domain.

Chaslot, Winands, and van den Herik [114] compared all three of these parallelization methods. They created a measure of scalability which was based on the increase in winning rates as the amount of time per turn increased, naming it the "strength speedup" measure. They concluded that the best increase in strength was achieved by using root parallelization, with average voting. This was tested on their Mango program. These experiments were conducted with reasonably sized tournaments from 500 to 100 games, and used a 1 second time limit per turn as the baseline.

Soejima, Kishimoto, and Watanabe [119] explored root parallelization, comparing a majority voting system versus an average voting system, and found that majority voting was superior. They compared these over 9×9, 19×19 sized Go boards with both self play using Fuego and also against MoGo. Additionally they tested move selection on particular board problems against an "oracle" version of the software which was Fuego running for 80 seconds, which is 8 times their baseline move time of 10 seconds in other experiments, but for the move selection experiments they ranged from 1 to 64 seconds. Their tournaments were limited in size, having only 200 games in each resulting in large confidence intervals on their results.

Segal [153] found a theoretical upper bound for a multi-threaded model over a single-threaded one for MCTS as implemented in Fuego. He also found that tree parallel algorithms work well using the virtual loss technique, but stop scaling well at around 8 threads without virtual loss. This research did use 1000 game tournaments, but always used self-play, which tends to exaggerate differences. Additionally, the testing was done by giving a total amount of time to both the multi-thread and single thread implementations, so the actual time for the multi-threaded player was the time divided by the number of threads. Strength was then extrapolated from these results, with the multi-threaded player always losing some Elo due to threading costs.

Schaefers and Platzner [130] analyzed the effects of using a parallel transposition table for tree parallelization along with dedicating some compute nodes to broadcast operations to help scaling to large numbers of nodes.

6.2 Multiple Algorithms for Root Parallelization

Along with comparing parallelization techniques, we introduce a new variant of root parallelization involving multiple algorithms. During the process of root parallelization, the different trees which are created grow slightly differently due to the stochastic nature of the playouts. Usually each tree is started with a different random seed for the stochastic parts of the algorithm to guarantee that the trees are built with different playout results. In order to increase the diversity of these trees, we introduce the idea of varying some of the tree-building parameters so that groups of the parallel trees will be created by slightly different build algorithms. For example, half of the parallel trees could use algorithm A while another half uses algorithm B. Our conjecture is that this diversity will help improve the overall results in the majority voting as was accomplished in an intelligent agents context in [127].

6.3 Methodology

We compared the effects of root parallelization over a number of scenarios in the domains of the games of Go and Hex. We chose to use the Go game domain because the MCTS software for computer Go has become quite mature, and Hex as a similar domain but with slightly different characteristics. The game of Go was the first to seriously develop MCTS, and offers a selection of open-source programs with which to experiment, including two that are considered to be some of the strongest computer Go programs.

We examined 4 different basic arrangements in order to observe the effects of parallelization on MCTS:

1. We increased the time available to a single thread, thereby allowing a larger tree to be created.
2. We increased the number of threads operating on a single tree to be increased, also allowing a larger tree to be created.
3. We increased the number of nodes over which to run MCTS, and therefore the number of trees that would be built and contribute to the choice of action.
4. We compared the effects of different settings for the MCTS algorithm using root

level parallelism by varying a parameter that controls the balance of of exploitation versus exploration.

6.3.1 Larger Trees With More Time

The longer an MCTS algorithm is run, the larger the search tree will become and this should usually improve the quality of the results. All trees eventually run into memory limitations, and have to have their very least promising branches pruned to free up the memory to build out the higher quality parts of the tree. Nonetheless, measured by the total number of playouts conducted, the size of the tree explored at some point, even if it has been subsequently pruned, rises close to linearly with the time allotted.

We allowed an MCTS program to run for longer than a typical amount of time in the domain, doubling and redoubling the time allowed, in order to see the effect this would have on the quality of the resultant decisions. We measured the quality by the comparing the winning rate of a computer Go program playing against an opponent that was only allowed to use the standard amount of time.

We chose a standard time allotment for each domain as the baseline amount of time. In the case of computer Go, using 15 second turns for decisions is quite common. We picked this as 1 unit of time, and then kept doubling the time allowed based on that unit. We believe this avoids the problems of showing large improvements from an unrealistically small baseline time unit. Similarly in the Hex domain we chose a 10 second per turn time limit as 1 unit of time.

6.3.2 Larger Trees With More Threads

Another way to increase the size of the search tree is to allow more than one thread to work on the same tree at a time: tree parallelization. The scalability of this approach is limited by the number of cores that are sharing access to the same memory on a machine with a shared memory architecture. Current high-end consumer-grade processors are four-core, but act as though they have eight cores by using hyper-threading technology.

We ran an MCTS program capable of conducting lock-free tree parallelization on a single machine with an increasing number of threads and recorded the quality of the win rate results as the number of threads increased.

6.3.3 Using More Trees

We increased the number of nodes over which to run MCTS, and therefore the number of trees that would be built and contribute to the choice of action. We implemented a parallel version of the MCTS algorithm with root level parallelism. Because of the success seen in [119], we used a system of majority voting to determine the next action to be taken.

At initialization, N copies of the MCTS program are created on different nodes of a cluster. Each node keeps track of the current state. When asked to produce an action choice, the node runs an MCTS search without sharing information with the other nodes. When the decision has been reached, each node reports its decision to the master node, node 0, but does not implement that decision: it does not change the state. For each possible action, Node 0 tallies up the number of nodes on which that action was chosen. The action with the largest number of votes, one vote per node, is considered the winner. This decision is then reported back to each node, which implements that winning action and changes the state accordingly. At this point, each node should have a copy of the current state and be ready for the next request.

We ran this majority vote root-parallel version of MCTS using an increasing number of nodes and recorded the changes in the winning rates.

6.3.4 Mixing Search Styles with Multiple Algorithms

We compared the effects of different settings for MCTS using root level parallelism by varying a parameter that controls the balance of exploitation versus exploration. In the descent phase of each round of MCTS, if the normal UCT formula is followed the exploration term will be set at infinity for any action not yet explored, as explained above. This causes every legal action from any state to be taken at least one time before any further descent occurs below that node. If infinity is replaced by a “first play urgency” term then depending on that value the exploration term may end up small enough to allow especially promising child nodes to be exploited before their siblings have had even a single play. By setting this urgency term to a low number such as 1, we will encourage exploitation over exploration more than the default balance between the two. Since a large number of independent trees will be constructed in the root parallel

algorithm, the exploration of the various promising branches can be accomplished by the presence of the votes from the different trees, rather than having to consider everything in one tree.

In addition we considered a mix of the default and low urgency settings where out of n nodes, half would operate with the default urgency settings and half would operate with the low setting. This was based on heterogeneous multi-agent work found in Marcolino [128], where it was indicated that a mix of different agents voting for an action could produce results superior to even the best of those agents on their own. Agent voting is very similar to the majority voting for combining the separately created trees in root parallelism, so we wished to observe any effects this might have in the parallel MCTS.

6.4 Experimental Setup

In order to study the effects of parallelization on MCTS in general and eliminate domain-specific results, we elected to perform identical experiments over two domains: the game of Go and the game of Hex.

In the domain of computer Go, there are currently two highly ranked open source Go engines which incorporate MCTS, Pachi [101] and Fuego [150]. We used recent builds of both of these programs – Pachi 11 and Fuego revision 1983 – to conduct our experiments.

Our Go experiments consisted of running tournaments between these two programs and recording the win/loss rates. All games were played with no handicap and a komi of 7.5. In each tournament, each computer Go program played half of its games as black, and half as white. We used Fuego and our modifications of Fuego as the variable program, modifying it and its parameters. These various versions of Fuego were then played against Pachi running always with the same set of parameters. All of our results are stated in terms of the percentage of games won by Fuego against Pachi, or won by a modified Fuego against its unmodified self. Pachi was played with all default parameters except that it was run as a single thread with a time limit of 15 seconds per turn with no opening book. Pachi plays with a RAVE component. Fuego was run with no opening book, at the time limit specified in the experiment, on the number of threads specified in the experiment and with the first-player-urgency specified in the experiment. Fuego

runs with a RAVE component and a low urgency setting of 1.

In order to observe the effects of parallelization in a situation where AMAF was not applicable, we also ran experiments with a version of Fuego which did not use the RAVE statistics. This version was run with a high urgency of 10,000 when not otherwise specified in the experiment.

In the domain of Hex, we used the two open-source programs MoHex and Wolve from the "benzene" [154] package of Hex programs developed at the University of Alberta. We built the software based on commit 7b1f7f from 16 February 2012. MoHex uses MCTS while Wolve is based on traditional $\alpha - \beta$ search. We used Wolve as our test opponent, and Mohex along with a modified MoHex as the program to test out the various kinds of parallelization. As with our experiments in the Go domain, all of our results are stated as the win rate of the MoHex program in its various versions versus the Wolve program.

The tournaments were run on a cluster of HP blade servers, where each node contained 2 quad-core 2.8 GHz Xeon "Nehalem EP" processors sharing 24 GB of memory.

In the game of Go two different board sizes are used to play games, especially between computers. A game on a 9×9 board usually lasts on average about 60 moves before it is decided or one side resigns. A game on a 19×19 board lasts about 260 moves, and so involves a much larger state space and takes from four to five times as long to play.

In the Hex domain, we used the common 11×11 board for all of our tournaments. On this size board, the typical game lasts on average a little less than 60 moves.

For the more time experiment in Go, we played a tournament on a 9×9 board with Fuego running a single thread for 15 seconds available per move against Pachi running a single thread for 15 seconds per turn. We then repeated this tournament five more times, allowing Fuego 30 seconds per turn, then 60, 120, 240, and finally 480 seconds per turn. We then repeated this experiment with the no RAVE version of Fuego.

In the Hex domain we played a tournament of 5 rounds with MoHex running on a single thread for 10 seconds per turn against Wolve at 10 seconds per turn. We then increased MoHex's allowed running time to 20, 40, 80, and 160 seconds per turn.

For the more threads experiment, we played four 9×9 tournaments matching up Fuego against Pachi, allowing Fuego to play with 1, 2, 4, and 8 threads respectively. Pachi was kept to a single thread. Both programs were given 15 second per turn. We

also ran these four tournaments with the no RAVE version of Fuego against Pachi. In the Hex domain we played MoHex with 1, 2, 4, and 8 threads against Wolve, all at 10 seconds per turn.

In the experiment with multiple trees in Go we played four tournaments on a 9×9 board. In each tournament both Fuego and Pachi were run on single threads with 15 second time limits. The number of nodes Fuego was allowed to use, and hence the number of trees it built, was varied with 1, 8, 32, and 64 nodes respectively in the tournaments. We also conducted another four tournaments with the opponent being Fuego itself running on a single thread and limited to one node. Again, the multi-tree Fuego was run with 1, 8, 32, and 64 nodes. Additionally we repeated these Fuego and Pachi tournaments but with the games played on a 19×19 board, single-threaded, with 15 second per turn time limits.

Experimenting with multiple trees in Hex we also played a similar four tournaments using 1, 8, 32, and 64 nodes for the MoHex player. All these tournaments were played with the 10 second per turn time limit against the Wolve player.

Finally, for the experiment mixing different styles of search, we used the variable number of nodes creating multiple trees experiment as a baseline, and then ran the same four tournaments against Pachi, but with a copy of Fuego where the first-move-urgency value had been set to 1 from its default value of 10,000. The default of 10,000 is a number representing a virtual infinite value for this parameter. Changing this value to 1 should encourage exploitation over exploration in the in-tree selection. In addition, we ran three tournaments where half of the nodes used the default urgency value, and half used the low urgency value. These were run on the 8, 32, and 64 node configurations. This experiment was run twice, with both the RAVE and no RAVE versions of Fuego.

6.4.1 Statistics

In order to study the trend of performance in the winning rates in an adversarial game situation we use the binomial confidence interval $p \pm 1.96\sqrt{p(1-p)/n}$ where p is the probability of a win and n is the number of trials in the sample. This means that 95% of the time an experiment is run the actual value sought will be within the confidence interval of the value seen in the sample by the experiment. Given this formula a reasonably large number of games must be played to detect an improvement in the software's

ability. If the winning rate is near 50% for example, the confidence interval for 500 games is a bit below 5%. This means with a 400 game trial, a result could be something like $50 \pm 5\%$. With an 800 game trial the confidence interval is near $\pm 3.5\%$.

Where some of the previous studies used tournament sizes in the hundreds, we chose to run tournaments of 1000 games in order to get confidence intervals close to $\pm 3\%$. We believe this level of precision is necessary to be certain that the observed results are real, not accidental, and to give a more accurate picture of the data trends.

6.5 Results

Tournaments between the computer Go opponents were moderated by the "gogui-twogtp" script [152] which communicates with each program using the simple Go Text Protocol and records the results of the games played. Tournaments between the computer Hex opponents were moderated by a similar script modified for Hex players.

6.5.1 Extended Time Results

The results from the tournaments between copies of Fuego with increasing amounts of time allotted for each decision and Pachi with the fixed 15 seconds of time per turn are shown in Table 6.1. As the amount of time allowed increases, the winning rate of Fuego increases until it is playing with eight times the base time, i.e., 120 seconds, after which no significant advantage is seen. The results from the no RAVE Fuego with increasing time limits show essentially no gain from increasing the time available when it is using the default urgency value.

Table 6.1: Fuego vs Pachi. Fuego win rates with increasing time/move limits for a 9×9 Go board.

Time	15 s	30 s	60 s	120 s	240 s	480 s
Winning Rate Fuego vs Pachi	48.4%	57.5%	63.8%	70.3%	68.6%	71.2%
Winning Rate Fuego (no RAVE) vs Pachi	8.3%	9.5%	10.3%	8.6%	9.4%	8.5%

The results from the tournaments between a copy of MoHex with varied and increasing time limits per turn are shown in Table 6.2. As the amount of time increases, the winning rate of Mohex continues to increase up through 8× where it appears to leveling out as it is doubled again to 16×.

Table 6.2: MoHex vs Wolve. MoHex win rates with increasing time/move limits on an 11×11 Hex board.

Time	10 s	20 s	40 s	80 s	160 s
Winning Rate MoHex vs Wolve	37.3%	42.8%	48.1%	54.6%	55.5%

6.5.2 Single Tree Multithreaded Results

Tournaments between copies of Fuego set to run with varying number of threads versus Pachi on a single thread produced the results seen in Table 6.3. As the number of threads increases, the strength of the program appears to increase almost linearly up to the maximum of eight, the number of processors in our shared-memory systems. Good results are also obtained for the no RAVE version of Fuego.

Table 6.3: Fuego win rates with increasing number of threads for a 9×9 Go board.

Number of Threads	1	2	4	8
Winning Rate Fuego vs Pachi	48.4%	59.2%	66.4%	74.2%
Winning Rate Fuego (no RAVE) vs Pachi	8.3%	10.5%	12.5%	18.9%

Results from a multi-threaded version of MoHex playing against Wolve are shown in Table 6.4. The improvement in the Hex domain appears much more muted than that gained in the Go domain.

6.5.3 Root Parallelization Results

Tournaments played between a root parallelized Fuego with an increasing number of nodes against single-threaded Pachi and the no RAVE version of Fuego against Pachi

Table 6.4: MoHex vs Wolve. MoHex win rates with increasing number of threads on an 11×11 Hex board.

Number of Threads	1	2	4	8
Winning Rate MoHex vs Wolve	37.9%	39.0%	43.6%	40.9%

are shown in Table 6.5. In both of these cases, the major jump in strength is from one to eight nodes.

Table 6.5: Fuego vs Pachi. Fuego win rates with increasing number of nodes for a 9×9 Go board.

Number of Nodes	1	8	32	64
Winning Rate Fuego vs Pachi	48.4%	62.8%	69.0%	68.5%
Winning Rate Fuego (no RAVE) vs Pachi	8.3%	22.8%	28.3%	23.4%

We ran the root parallel version of MoHex on an increasing number of nodes versus the default Wolve program. This was done once with the standard 10 second per turn time limit and once again with a 20 second per turn time limit. The results from these tournaments are shown in Table 6.6.

Table 6.6: MoHex vs Wolve. MoHex win rates with increasing number of nodes on an 11×11 Hex board.

Number of Nodes	1	8	16	32	64
10s/turn Winning Rate MoHex vs Wolve	37.3%	40.8%	40.7%	38.2%	41.5%
20s/turn Winning Rate MoHex vs Wolve	42.8%	41.8%	43.5%	44.3%	45.7%

The root parallel version of Fuego played against a single-threaded copy of Fuego with the results shown in Table 6.7. When competing against itself, the parallelized

Fuego seemed to hit its upper limit at a lower winning rate than when competing against Pachi. Self-play is typically not used in testing programs because implementation side effects can confound the results.

Table 6.7: Fuego vs Fuego. Fuego win rates with increasing number of nodes for a 9×9 Go board.

Number of Nodes	1	8	32	64
Winning Rate	49.8%	63.0%	62.8%	63.9%

We also ran a set of tournaments on a larger size Go board, 19×19. These were not extended to 1000 games per tournament, and so do not have as much precision as our other results, but show the same trend over an increasing number of nodes as the 9×9 results: a major leap in strength from 1 to 8 nodes, and then only gradual gains after that. These results are shown in Table 6.8.

Table 6.8: Fuego vs Pachi on a 19×19 Go board. Fuego win rates with increasing number of nodes.

Number of Nodes	1	8	32	64	128
Winning Rate Fuego vs Pachi	40.9%	61.3%	63.1%	67.0%	66.7

Finally we ran the root parallel version of MoHex on an increasing number of nodes versus the default Wolve program. This was done once with the standard 10 second per turn time limit and once again with a 20 second per turn time limit. The results from these tournaments are shown in Table 6.9.

6.5.4 Comparing Parallelization Techniques

We compare the effects of these techniques in our Fuego vs Pachi tournaments in Figure 6.1. As the time or number of threads or number of trees increases, all of these techniques increase their strength from a baseline value. Increasing the number of threads using a lock-free implementation works the best, but is limited by the number of CPUs in a shared memory architecture. Increasing the number of nodes scales almost as well as

Table 6.9: MoHex vs Wolve. MoHex win rates with increasing number of nodes on an 11×11 Hex board.

Number of Nodes	1	8	16	32	64
10s/turn Winning Rate MoHex vs Wolve	37.3%	40.8%	40.7%	38.2%	41.5%
20s/turn Winning Rate MoHex vs Wolve	42.8%	41.8%	43.5%	44.3%	45.7%

increasing the time, but both of these see diminishing returns between the 8 and 32 multiplier levels.

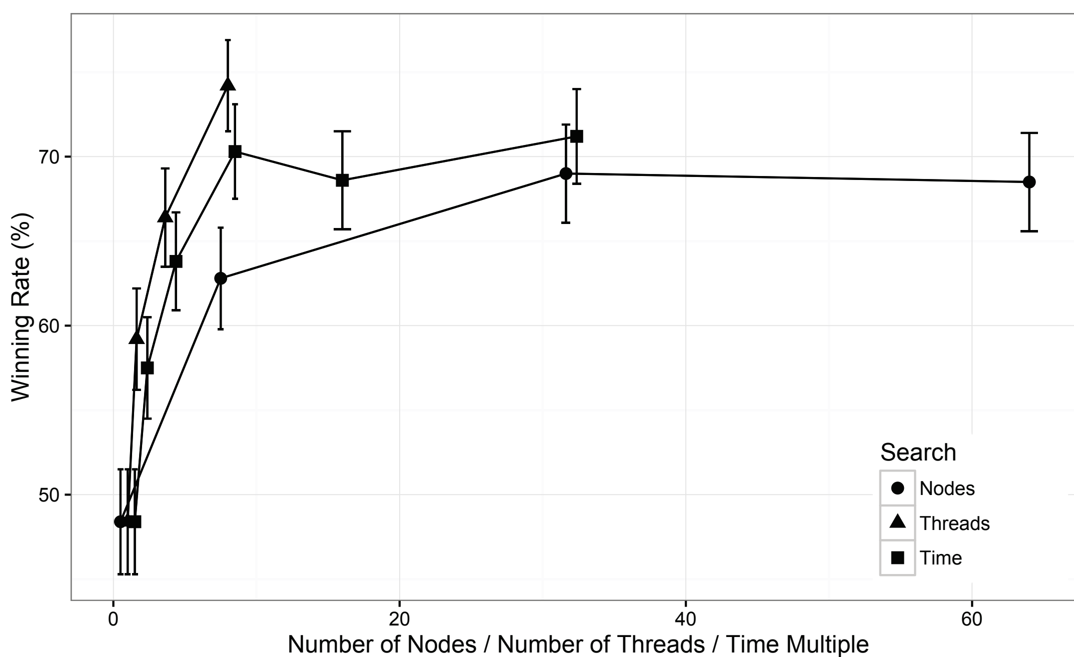


Figure 6.1: Comparison of winning rates for root parallelization, multi-threaded, and increased time MCTS for a 9×9 Go board.

We compare these effects again in our Fuego (no RAVE) vs Pachi tournaments shown in Figure 6.2. In this case the best result is achieved by increasing the number of nodes rather than the number of threads, but appears to maximize at 32 nodes.

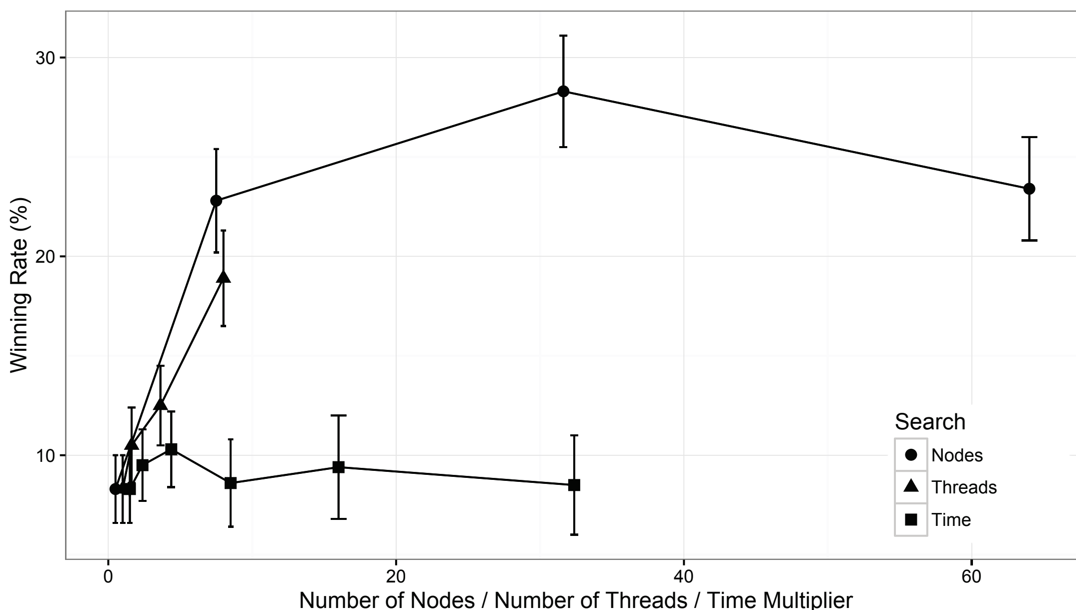


Figure 6.2: Comparison of Fuego (no RAVE) winning rates vs Pachi for root parallelization, multi-threaded, and increased time MCTS for a 9×9 Go board.

The effects of increasing the time and increasing the number of nodes across both 10 second per turn and 20 second per turn MoHex on winning rates against 10 second per turn Wolve are shown in Figure 6.3. This clearly shows that increasing from 1 to 64 trees with root parallelization produces very little gain compared to increasing the amount of time spent building a single tree in the domain of Hex.

We also compare the winning rates of our root parallel version of Fuego on the different board sizes: 9×9 against 19×19. The winning rates given the increasing number of nodes is shown in Figure 6.4. This shows that the win rates stop increasing after 32 nodes on the smaller board, but continue increasing up to 64 nodes on the larger board.

To investigate the apparent difference made by the size of the domain on the scalability of root parallelism, we have started to examine the details of the voting patterns with differing numbers of nodes.

We looked at how many different actions were chosen by the independent compute nodes in the root parallel process at any decision point, and how that varied as the

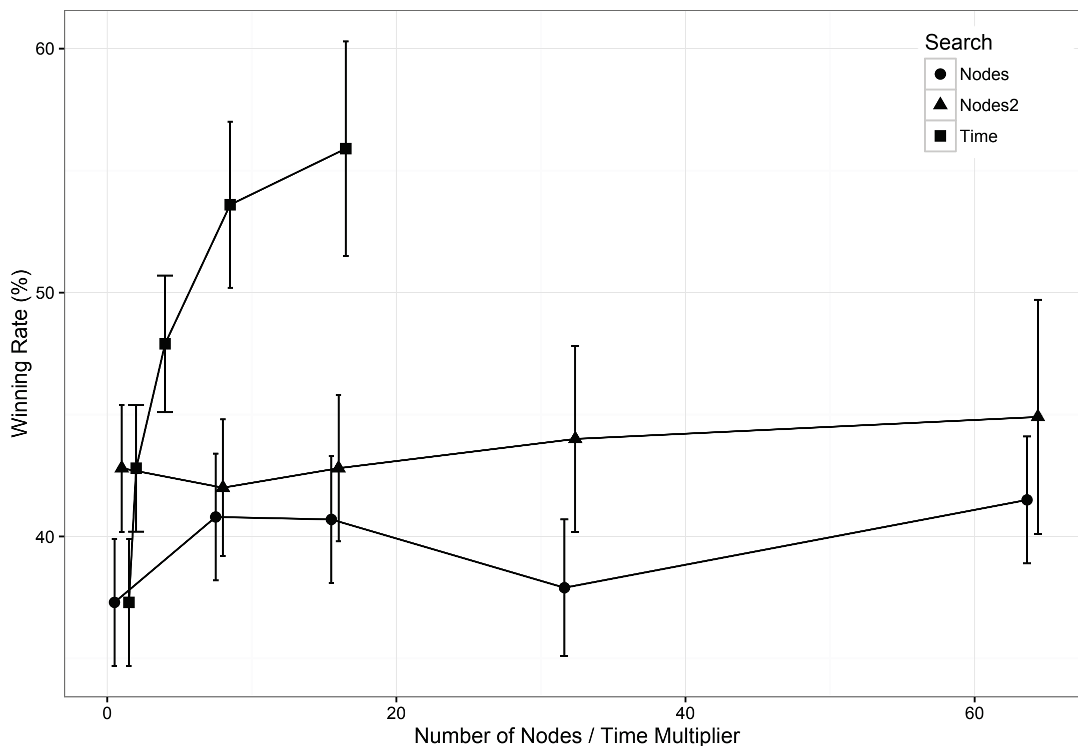


Figure 6.3: Comparison of winning rates for root parallelization at two timing levels, along with increased time and increased threads on an 11×11 Hex board.

number of compute nodes increased. To do so we looked at a 5 randomly selected games and for each turn we recorded how many different actions received any votes, and how many votes they received. This value was then averaged across all the moves of all the games to arrive at number for each board size and number of compute nodes combination.

The results are shown in Figure 6.5 and seem to correlate well with the winning rate results: in the 19×19 domain more reasonable possible moves are available than in 9×9, and increase as the number of compute nodes increases. The number of moves for the 9×9, however, plateaus at 4 choices at 64 compute nodes, very likely because there are simply not more than 4 reasonable choices available in the smaller domain.

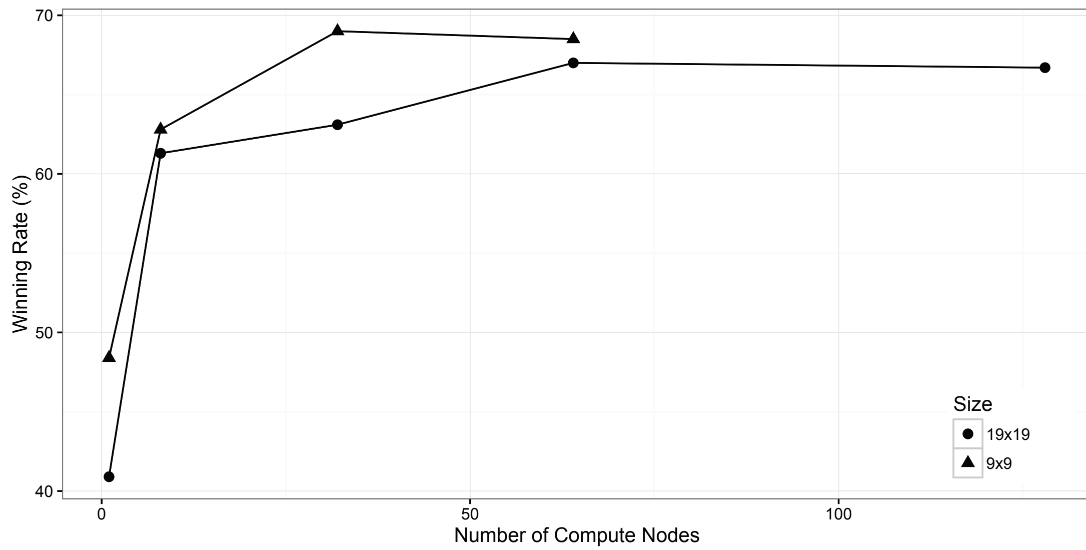


Figure 6.4: Comparison of winning rates for root parallelization of Go at two board sizes.

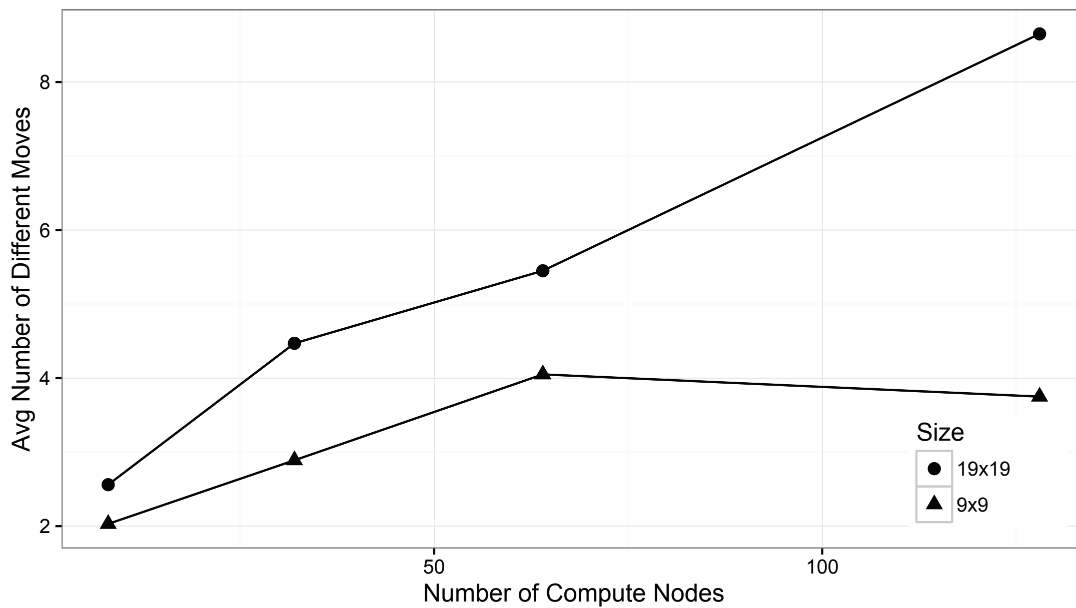


Figure 6.5: Average number of different moves chosen during root parallelization in Go at two board sizes.

6.5.5 Mixed Search Results

We compared the results of mixing algorithms, in this case the first play urgency value which replaces infinity in the UCT formula, as the number of nodes available to the root-parallel MCTS increased. The default urgency value is 10,000 while the low urgency we used is 1. These win rates are shown in Table 6.10. In the case of modifying just the first play urgency, there were no statistically significant differences in performance compared to the default values.

Table 6.10: Fuego vs Pachi. Fuego win rates with increasing number of nodes for a 9×9 Go board.

Urgency	1	8	32	64
Default	48.4%	62.8%	69.0%	68.5%
Low	48.4%	64.7%	67.3%	68.3%
Mixed		65.6%	65.0%	68.7%

This experiment was repeated with the no RAVE version of Fuego. The results of this are seen in Table 6.11, and also fail to show a statistically significant difference between playing with default, low, or mixed values of the first play urgency setting.

Table 6.11: Fuego no RAVE vs Pachi. Fuego win rates with increasing number of nodes for a 9×9 Go board.

Urgency	1	8	32	64
Default	8.3%	22.8%	23.3%	23.4%
Low	9.2%	24.6%	27.0%	25.5%
Mixed		24.6%	27.3%	27.5%

6.6 Analysis of Results

Compared to previous surveys found in the literature which have found that parallel algorithms underperformed against simply running the algorithm for a longer period of time in the domain of Go, we have shown that parallel algorithms keep pace with or

exceed the performance gained by increasing the amount of time. We believe this is due to both the maturity of the algorithms now available, and the use of a reasonable baseline time unit for measurement. Weaker programs have more variability in their playing strength, and it is deceptively easy to get a jump in performance when starting from an unrealisticly low time scale. On the other hand, our results in the Hex domain concur with these previous studies that increasing time, if available, is the best way to improve winning rates.

Our larger tournament sizes also provide a much better picture of the actual results and trend lines due to the smaller confidence intervals compared to many of the previous studies, with which most of our results do agree.

In the Hex domain, however, root parallelism did not keep pace with the improvements that could be gained by increasing the amount of time available to create a single tree. Indeed increasing the number of nodes to 64 appeared to have about the same effect as doubling the amount of time, both from 10 second turns to 20 second turns, and from 20 second turns to 40 second turns.

One interesting result was that although in 9×9 Go and in Hex root parallelization gains stalled at 32 nodes, in 19×19 Go the gains continued through 64 nodes. By examining the number of different choices made in actual play in a root parallel system, we believe that the plateau is due to the larger number of good choices available in the larger game. As the number of votes available increases the likelihood of picking the best move increases, but this may plateau at some multiple of the available good moves. For example, if there are only three reasonable moves in a 9×9 game having 64 instead of 32 voters choosing may not increase the quality of the pick, but if there are ten reasonable moves available it may require 64 or more nodes to find one of the best as many of the votes will be spread out over the wider selection of reasonable moves. We believe that our observations of the vote distribution support this theory, and will include more studies of this nature in our future research.

The lack of a statistically significant difference between the default urgency, low urgency, and mixed urgency algorithms was disappointing. This is likely due to the efficiency of the RAVE statistics (all moves as first: AMAF) which allow non-infinite values to be attributed to actions which have not yet been expanded from a given node, thus bypassing the use of the urgency parameter. Because AMAF is not valid in many

domains, though it is in Go, this experiment should be repeated and expanded in a different domain, or in Go but without using RAVE.

Chapter 7

Conclusions and Future Work

Artificial Intelligence researchers have for decades applied their skills to games, in particular to Othello, chess, checkers, and Go. Automatically playing these games is challenging due to their large size of the possible moves search space and the lack of heuristics to guide the search. Recently, Monte Carlo methods, particularly Monte Carlo Tree Search, MCTS, has proven to be the key that unlocks the door to progress, notable in Go, the domain of this thesis.

7.1 Conclusions

In this work we have addressed two important aspects of MCTS for Go.

First, we approached the opening game, which we improved via a novel form of an opening book, one which informs the processing of MCTS. We introduced SmartStart, a method which improves Monte Carlo search at the beginning of the game of Go, where its search tree is at its widest and deepest. This method uses expert knowledge to eliminate from consideration moves which have not been played by professional Go players in similar situations. These moves have been taken from publically available sources.

SmartStart creates a multi-element representation for each board position and then matches that against clusters of professional games. A unique idea in SmartStart is that instead of searching through individual games in a database in a brute force way, during pre-processing it clusters these games and then searches only by cluster center

positions. SmartStart allows only those next moves which were played in games in the closest cluster to be searched by the Monte Carlo algorithm. The benefit of this approach is that it speeds up processing, working as a very fast filter. By pruning a large proportion of the options at the very beginning of a game tree, stochastic search can spend its time on the most fruitful move possibilities. Applying this technique has raised the win rate of a Monte Carlo program by a modest, but statistically significant amount.

Second, we have studied the parallelization of MCST for Go, and another game, Hex, and have introduced a new way to parallelize the search.

Although increasing the amount of time to build a tree using MCTS increases the quality of the results, this may in some cases be impractical or disadvantageous.

Previous studies comparing parallelization schemes have suffered from either using too few trials to achieve a reasonable statistical significance, or have used indirect measures of the effectiveness. We compared the root and tree methods of parallelizing against increasing the amount of time available using the winning rate of the programs as the efficacy measure.We found that the using either root or tree parallelization with current open source software compared well to previous results, in that both were able to keep pace with and outperform the improvements gained by increasing the amount of time. This was an improvement on results presented previously, which we believe can be attributed to the maturation of the software and also our measurements methods, including using a reasonable baseline unit of time.

We introduced a new variant of root parallelization. It utilizes multiple multiple parameter sets in order to increase the diversity of the separate trees constructed. This is similar to the idea of multiple agents voting on the same move as in the agents environment, and found that although the performance did not show a significant advantage with only parameter sets, it did not decrease performance, and so this method still shows some promise as a way to better utilize parallelization.

7.2 Future Work

As with any research, this work has created some new research questions.

7.2.1 SmartStart

In the opening game, would increasing the number of clusters as the move number increases better capture the greater variability of the go games at these later moves, as the very opening moves transition into the late opening and early middle game?

Would a combination of SmartStart and more traditional opening book techniques produce better results? In particular, a combination of Baier's technique of first checking for exact match on the whole board, then in the corners, and then moving on to SmartStart.

Could SmartStart's performance be improved further by varying the bias applied based on the relative popularity of the move within the cluster: more popular means more bias? One possible way of doing this might be to have a fixed number of bias points in a pool which are then distributed exactly as the next-moves are distributed within the matched cluster.

7.2.2 Parallelization

What are the effects of combining root parallelization with tree parallelization? Will having multiple threads running on each node in a cluster lower the performance gains as the number of nodes increases?

Choose other and more parameters to vary with heterogeneous root parallelism. Search for a parameter which really allows different trees to be created. Is this variability any better than the variability achieved with the random seed?

What is the reason for the better performance scaling of the root parallel algorithm in 19×19 Go compared to 9×9 Go? We propose to further examine the distribution of the choices made in these root parallel systems. Given 8 through 128 choices, what is the nature of the distribution of chosen moves in 19×19 Go versus 9×9 Go; the smaller versus the larger domain? Does the distribution change as the game progresses?

What are the perfomance gains from parallelization in a different domain which is not a perfect-information game, and not a RAVE domain?

References

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 01 2016.

[2] Shirakawa Masayoshi. *A Journey in Search of the Origins of Go*. Yutopian Enterprises, 2005.

[3] Daniel James Edwards and T.P. Hart. The alpha-beta heuristic. Technical Report AIM-030, MIT, 1961.

[4] Bernd Brügmann. Monte Carlo Go. http://www.ideanest.com/vegos/MonteCarloGo.pdf, October 1993.

[5] Haruhiro Yoshimoto, Kazuki Yoshizoe, Tomoyuki Kaneko, Akihiro Kishimoto, and Kenjiro Taura. Monte Carlo Go has a way to go. In *21st Nat'l Conf. on Artificial Intelligence (AAAI-06)*, pages 1070–1075, 2006.

[6] Cameron Browne. The dangers of random playouts. *ICGA Journal*, 34(1):25–26, 2011.

[7] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[8] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine*

Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[9] Peter Drake. The last-good-reply policy for Monte-Carlo Go. *International Computer Games Association Journal*, 32(4):221–227, 2009.

[10] Hendrik Baier and P. D. Drake. The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go. *Computational Intelligence and AI in Games, IEEE Trans. on*, 2(4):303–309, Dec 2010.

[11] Kamil Rocki and Reiji Suda. Large-scale parallel Monte Carlo tree search on GPU. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, pages 2034–2037, 2011.

[12] Markus Enzenberger and Martin Müller. A lock-free multithreaded Monte-Carlo Tree Search algorithm. In *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*, pages 14–20, 2009.

[13] A. L. Brudno. Bounds and valuations for shortening the search of estimates. *Problems of Cybernetics*, pages 225–241, 1963.

[14] John Tromp and Gunnar Farnebäck. Combinatorics of Go. January 2007.

[15] Thomas Wolf and Lei Shen. Checking life-and-death problems in go i: The program scanld. *ICGA Journal*, 30(2):67–74, 2007.

[16] Bruno Bouzy and Tristan Cazenave. Computer Go: an AI Oriented Survey. *Artificial Intelligence*, 132:39–103, 2001.

[17] Martin Müller. Computer Go. *Artificial Intelligence*, 134:145–179, 2002.

[18] Erik van der Werf. *AI techniques for the game of Go.* PhD thesis, Universitaire Pers Maastricht, 2004.

[19] Daniel Bump and Gunnar Farnebäck. Gnugo. http://www.gnugo.org/software/gnugo, 2008.

[20] J.L. Ryder. *Heuristic Analysis of Large Trees as Generated in the Game of Go*. PhD thesis, Stanford University, 1971.

[21] Albert L. Zobrist. *Feature Extraction and Representation for Pattern Recognition and the Game of Go*. PhD thesis, University of Wisconsin, 1970.

[22] Tristan Cazenave. Automatic acquisition of tactical Go rules. In *3rd Game Programming Workshop in Japan*, Hakone, 1996.

[23] W. Reitman and B. Wilcox. The Structure and Performance of the interim.2 Go Program. In *Proceedings of the 6th IJCAI*, pages 711–719, 1979.

[24] Xindi Cai and D.C. II Wunsch. A parallel computer-Go player, using HDP method. In *IJCNN '01. International Joint Conference on Neural Networks.*, volume 4, pages 2373–2375, 2001.

[25] David B. Benson. Life in the game of go. *Inf. Sci.*, 10(1):17–29, 1976.

[26] Jay Burmeister and Janet Wiles. The challenge of go as a domain for ai research: a comparison between go and chess. In *Intelligent Information Systems, 1995. ANZIIS-95. Proceedings of the Third Australian and New Zealand Conference on*, pages 181–186. IEEE, 1995.

[27] Jay Burmeister, Janet Wiles, and Helen Purchase. The integration of cognitive knowledge into perceptual representations in computer go. In *Proceedings of the Second Game Programming Workshop in Japan*, 1995.

[28] Jay Burmeister, Janet Wiles, and Helen Purchase. On relating local and global factors: a case study from the game of go. In *Intelligent Information Systems, 1995. ANZIIS-95. Proceedings of the Third Australian and New Zealand Conference on*, pages 187–192. IEEE, 1995.

[29] Jay Burmeister and Janet Wiles. The use of inferential information in remembering go positions. In *Proceedings of the Third Game Programming Workshop in Japan, Hakone*, pages 56–65, 1996.

[30] Jay Madison Burmeister. *Studies in human and computer Go: assessing the game of Go as a research domain for cognitive science*. PhD thesis, University of Queensland, 2000.

[31] Bruno Bouzy. Spatial reasoning in the game of go. In *Workshop on Representations and Processes in Vision and Natural Language, ECAI*, pages 78–80, 1996.

[32] Bruno Bouzy. Mathematical morphology applied to computer go. *IJPRAI*, 17(2):257–268, 2003.

[33] Bruno Bouzy and Tristan Cazenave. Shared concepts between complex domains and the game of Go. Technical Report LAFORIAIBP, Université Pierre et Marie Curie, Paris, 1996.

[34] ER Berlekamp. The economists view of combinatorial games. *Games of No Chance*, pages 365–405, 1996.

[35] Martin Müller, Elwyn Berlekamp, and Bill Spight. Generalized thermography: Algorithms, implementation, and application to go endgames. citeseerx.ist.psu.edu, 1996. doi=10.1.1.34.6699.

[36] ER Berlekamp and Yonghoan Kim. Where is the thousand-dollar ko?. *Games of No Chance*, pages 203–226, 1996.

[37] William L. Spight. Extended thermography for multiple kos in go. *Theor. Comput. Sci.*, 252(1-2):23–43, 2001.

[38] Martin Müller, Markus Enzenberger, and Jonathan Schaeffer. Temperature discovery search. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 658–663. AAAI Press / The MIT Press, 2004.

[39] David Silver, Richard S. Sutton, and Martin Müller. Temporal-difference search in computer Go. *Machine Learning*, 87(2):183–219, 2012.

[40] Kuo-Yuan Kao. Mean and temperature search for go endgames. *Inf. Sci.*, 122(1):77–90, 2000.

[41] Jörg Bewersdorff. Go und Mathematik, 1998.

[42] Ken Chen and Zhixing Chen. Static analysis of life and death in the game of go. *Information Sciences*, 121(12):113 – 134, 1999.

[43] David Wolfe. Go endgames are pspace-hard. *More Games of No Chance*, 42:125–136, 2002.

[44] Antti Huima. A group-theoretic zobrist hash function, 2000.

[45] Thore Graepel, Mike Goutri, Marco Kruger, and Ralf Herbrich. Go, svm, go. citeseerx.ist.psu.edu, 2000. doi=10.1.1.34.5745.

[46] Thomas Wolf. Forward pruning and other heuristic search techniques in tsume go. *Inf. Sci.*, 122(1):59–76, 2000.

[47] Thomas Wolf. Two applications of a life & death problem solver in go. *Journal of ÖGAI*, 26(2):11–18, 2007.

[48] Keh-Hsun Chen. Computer go: Knowledge, search, and move decision. *ICGA Journal*, 24(4):203–215, 2001.

[49] Richard Cant, Julian Churchill, and David Al-Dabass. Using hard and soft artificial intelligence algorithms to simulate human Go playing techniques. *International Journal of Simulation Systems, Science & Technology*, 2(1):31–49, June 2001.

[50] Bruno Bouzy. The move-decision strategy of indigo. *ICGA Journal*, 26(1):14–27, 2003.

[51] Keh-Hsun Chen. Maximizing the chance of winning in searching go game trees. *Information Sciences*, 175(4):273 – 283, 2005. Heuristic Search and Computer Game Playing {IV}.

[52] Katsuhiko Nakamura. Static analysis based on formal models and incremental computation in go programming. *Theor. Comput. Sci.*, 349(2):184–201, 2005.

[53] Erik C. D. van der Werf, Mark H. M. Winands, H. Jaap van den Herik, and Jos W. H. M. Uiterwijk. Learning to predict life and death from go game records. *Inf. Sci.*, 175(4):258–272, 2005.

[54] Emil H. J. Nijhuis. Learning patterns in the game of go. Master's thesis, Universiteit van Amsterdam, 2006.

[55] Lin Wu and Pierre Baldi. A scalable machine learning approach to go. In Bernhard Schölkopf, John C. Platt, and Thomas Hofmann, editors, *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 1521–1528. MIT Press, 2006.

[56] Bruno Bouzy and Guillaume Chaslot. Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005*. IEEE, 2005.

[57] David Stern, Ralf Herbrich, and Thore Graepel. Bayesian pattern ranking for move prediction in the game of go. In *International Conference on Machine Learning (ICML-2006)*, 2006.

[58] David Silver, Richard S. Sutton, and Martin Müller. Reinforcement learning of local shape in the game of go. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 1053–1058, 2007.

[59] Helmut A. Mayer. Board representations for neural go players learning by temporal difference. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007, Honolulu, Hawaii, USA, 1-5 April, 2007*, pages 183–188. IEEE, 2007.

[60] Chang-Shing Lee, Mei-Hui Wang, Tzung-Pei Hong, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Yau-Hwang Kuo. A novel ontology for computer Go knowledge management. In *FUZZ-IEEE 2009, IEEE*

International Conference on Fuzzy Systems, Jeju Island, Korea, 20-24 August 2009, Proceedings, pages 1056–1061. IEEE, 2009.

[61] Martin Wistuba, Lars Schaefers, and Marco Platzner. Comparison of bayesian move prediction systems for computer go. In *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, Granada, Spain, September 11-14, 2012*, pages 91–99, 2012.

[62] Thomas Wolf. Basic seki in go. Technical report, technical report, Department of Mathematics, Brock University, 2012.

[63] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in Go using deep convolutional neural networks. *CoRR*, abs/1412.6564, 2014.

[64] Tim Huang, Graeme Connell, and Bryan McQuade. Experiments with learning opening strategy in the game of go. *International Journal on Artificial Intelligence Tools*, 13(1):101–104, 2004.

[65] Byung-Doo Lee, Hans Werner Guesgen, and Jacky Baltes. The application of TD (λ) learning to the opening games of 19×19 Go, 2003.

[66] Graham Kendall, Razali Yaakob, and Philip Hingston. An investigation of an evolutionary approach to the opening of go. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2004, 19-23 June 2004, Portland, OR, USA*, pages 2052–2059. IEEE, 2004.

[67] Pierre Audouard, Guillaume Chaslot, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Grid coevolution for adaptive simulations: Application to the building of opening books in the game of go. In *EvoWorkshops*, volume 5484 of *Lecture Notes in Computer Science*, pages 323–332. Springer, 2009.

[68] Romaric Gaudel, Jean-Baptiste Hoock, Julien Perez, Nataliya Sokolovska, and Olivier Teytaud. A principled method for exploiting opening books. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 136–144. Springer, 2010.

[69] Jessica Mullins and Peter Drake. Using human knowledge to improve opening strategy in computer go. In *Proc. Int'l Conf. on Artificial Intelligence (ICAI)*, pages 730–734, 2010.

[70] Jean-Baptiste Hoock, Chang-Shing Lee, Arpad Rimmel, Fabien Teytaud, Mei-Hui Wang, and Olivier Teytaud. Intelligent agents for the game of go. *IEEE Comp. Int. Mag.*, 5(4):28–42, 2010.

[71] Hendrik Baier and Mark H. M. Winands. Active opening book application for Monte-Carlo Tree Search in 19x19 Go. In Patrick De Causmaecker, editor, *Proc. 23rd Benelux Conference on Artificial Intelligence*, pages 3–10, 2011.

[72] Martin Michalowski, Mark Boddy, and Mike Neilsen. Bayesian learning of generalized board positions for improved move prediction in computer Go. In *Proc. 25th Conf. on Artificial Intelligence (AAAI)*, 2011.

[73] Kwangjin Hong, Jinuk Kim, Hongwon Lee, Jihoon Lee, Jiwoong Heo, Sunchul Kim, and Keechul Jung. Data-driven computer go based on hadoop. In Leonard Barolli, Kin Fun Li, Tomoya Enokido, Fatos Xhafa, and Makoto Takizawa, editors, *28th International Conference on Advanced Information Networking and Applications Workshops, AINA 2014 Workshops, Victoria, BC, Canada, May 13-16, 2014*, pages 347–351. IEEE Computer Society, 2014.

[74] Erik S. Steinmetz and Maria L. Gini. Mining expert play to guide monte carlo search in the opening moves of go. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 801–807. AAAI Press, 2015.

[75] Bruno Bouzy and Bernard Helmstetter. Monte-carlo go developments. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Advances in Computer Games, Many Games, Many Challenges, 10th International Conference, ACG 2003, Graz, Austria, November 24-27, 2003, Revised Papers*, volume 263 of *IFIP*, pages 159–174. Kluwer, 2003.

[76] Bruno Bouzy and B. Helmstetter. Monte-Carlo Go Developments. In Ernst A. Heinz, H. Jaap van den Herik, and Hiroyuki Iida, editors, *Advances in Computer Games conference (ACG-10), Graz 2003*. Klewer Academic Publishers, 2003.

[77] Bruno Bouzy. Associating shallow and selective global tree search with monte carlo for 9*9 go. In H. Jaap van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *Computers and Games, 4th International Conference, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 67–80. Springer, 2004.

[78] Bruno Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences, Heuristic Search and Computer Playing IV*, 175(4):247–257, 2005.

[79] Bruno Bouzy. Move-pruning techniques for monte-carlo go. In H. Jaap van den Herik, Shun-chin Hsu, Tsan-sheng Hsu, and H. H. L. M. Donkers, editors, *Advances in Computer Games, 11th International Conference, ACG 2005, Taipei, Taiwan, September 6-9, 2005. Revised Papers*, volume 4250 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2006.

[80] GMJB Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, JWHM Uiterwijk, and H Jaap Van Den Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

[81] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006, Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[82] Rémi Coulom. Computing "elo ratings" of move patterns in the game of go. *ICGA Journal*, 30(4):198–208, 2007.

[83] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[84] Sylvain Gelly, Yizao Wang, Olivier Teytaud, Modification Uct Patterns, and Projet Tao. Modification of UCT with patterns in monte-carlo Go. citeseerx.ist.psu.edu, 2006. doi=10.1.1.96.7727.

[85] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[86] Peter Drake, Andrew Pouliot, Niku Schreiner, and Bjorn Vanberg. The proximity heuristic and an opening book in Monte Carlo Go. *AAAI*, 2007.

[87] Peter Drake and Steve Uurtamo. Heuristics in Monte Carlo Go. In *Proc. 2007 Int'l Conf. on Artificial Intelligence (IJCAI)*, 2007.

[88] Guillaume Chaslot, Louis Chatriot, C Fiter, Sylvain Gelly, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Combining expert, offline, transient and online knowledge in monte-carlo exploration. citeseerx.ist.psu.edu, 2008. doi=10.1.1.169.8073.

[89] Darren Cook. Computer-go mailing list. Speed go next thing to explore thread. http://computer-go.org/pipermail/computer-go/2007-April/009640.html, April 2007.

[90] Radha-Krishna Balla and Alan Fern. UCT for tactical assault planning in real-time strategy games. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 40–45, 2009.

[91] Diego Perez Liebana, Spyridon Samothrakis, and Simon M. Lucas. Online and offline learning in multi-objective monte carlo tree search. In *2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, August 11-13, 2013*, pages 1–8. IEEE, 2013.

[92] Keh-Hsun Chen, Dawei Du, and Peigang Zhang. Monte-carlo tree search and computer go. In *Advances in Information and Intelligent Systems*, pages 201–225, 2009.

[93] Arpad Rimmel, Olivier Teytaud, Chang-Shing Lee, Shi-Jim Yen, Mei-Hui Wang, and Shang-Rong Tsai. Current frontiers in computer Go. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):229–238, 2010.

[94] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer Go: Monte Carlo Tree Search and extensions. *Commun. ACM*, 55(3):106–113, 2012.

[95] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo Tree Search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, March 2012.

[96] Shih-Chieh Huang, Rémi Coulom, and Shun-Shii Lin. Monte-carlo simulation balancing in practice. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2010.

[97] Arpad Rimmel, Fabien Teytaud, and Olivier Teytaud. Biasing monte-carlo simulations through RAVE values. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2010.

[98] Sumudu Fernando and Martin Müller. Analyzing simulations in monte-carlo tree search for the game of go. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, volume 8427 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2013.

[99] Edward J. Powley, Daniel Whitehouse, and Peter I. Cowling. Bandits all the way down: UCB1 as a simulation policy in monte carlo tree search. In *2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, August 11-13, 2013*, pages 1–8. IEEE, 2013.

[100] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[101] Petr Baudiš and Jean-Loup Gailly. Pachi: State of the art open source Go program. In H. Jaap van den Herik and Aske Plaat, editors, *Advances in Computer Games - 13th International Conference, ACG 2011, Tilburg, The Netherlands, November 20-22, 2011, Revised Selected Papers*, volume 7168 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2011.

[102] Junichi Hashimoto, Akihiro Kishimoto, Kazuki Yoshizoe, and Kokolo Ikeda. Accelerated UCT and its application to two-player games. In H. Jaap van den Herik and Aske Plaat, editors, *Advances in Computer Games - 13th International Conference, ACG 2011, Tilburg, The Netherlands, November 20-22, 2011, Revised Selected Papers*, volume 7168 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2011.

[103] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 5(1):1–122, 2012.

[104] Sébastien Bubeck, Vianney Perchet, and Philippe Rigollet. Bounded regret in stochastic multi-armed bandits. In Shai Shalev-Shwartz and Ingo Steinwart, editors, *COLT 2013 - The 26th Annual Conference on Learning Theory, June 12-14, 2013, Princeton University, NJ, USA*, volume 30 of *JMLR Proceedings*, pages 122–134. JMLR.org, 2013.

[105] Francois van Niekerk and Steve Kroon. Decision trees for computer go features. In Tristan Cazenave, Mark H. M. Winands, and Hiroyuki Iida, editors, *Computer Games - Workshop on Computer Games, CGW 2013, Held in Conjunction with the 23rd International Conference on Artificial Intelligence, IJCAI 2013, Beijing,*

China, August 3, 2013, Revised Selected Papers, volume 408 of *Communications in Computer and Information Science*, pages 44–56. Springer, 2013.

[106] Kokolo Ikeda and Simon Viennot. Efficiency of static knowledge bias in monte-carlo tree search. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013. Revised Selected Papers*, volume 8427 of *Lecture Notes in Computer Science*, pages 26–38. Springer, 2013.

[107] Tobias Graf, Lars Schaefers, and Marco Platzner. On semeai detection in monte-carlo go. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, volume 8427 of *Lecture Notes in Computer Science*, pages 14–25. Springer, 2013.

[108] Shih-Chieh Huang and Martin Müller. Investigating the limits of monte-carlo tree search methods in computer go. In *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, pages 39–48, 2013.

[109] Richard Lorentz and Therese Horey. Programming breakthrough. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, volume 8427 of *Lecture Notes in Computer Science*, pages 49–59. Springer, 2013.

[110] Tobias Graf and Marco Platzner. Adaptive playouts in monte-carlo tree search with policy-gradient reinforcement learning. In Aske Plaat, H. Jaap van den Herik, and Walter A. Kosters, editors, *Advances in Computer Games - 14th International Conference, ACG 2015, Leiden, The Netherlands, July 1-3, 2015, Revised Selected Papers*, volume 9525 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2015.

[111] Peter H. Jin and Kurt Keutzer. Convolutional monte carlo rollouts in go. *CoRR*, abs/1512.03375, 2015.

[112] Tristan Cazenave and Nicolas Jouandeau. On the parallelization of uct. *Proceedings of CGW07*, pages 93–101, 2007.

[113] Tristan Cazenave and Nicolas Jouandeau. A parallel monte-carlo tree search algorithm. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 72–80. Springer, 2008.

[114] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. In *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008*, pages 60–71, 2008.

[115] Hideki Kato, Ikuo Takeuchi, et al. Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08)*, 2008.

[116] Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Y. Kalemkarian. The parallelization of monte-carlo planning - parallelization of mc-planning. In *ICINCO 2008, Proceedings of the Fifth International Conference on Informatics in Control, Automation and Robotics, Intelligent Control Systems and Optimization, Funchal, Madeira, Portugal, May 11-15, 2008*, pages 244–249, 2008.

[117] Jean Méhat and Tristan Cazenave. A parallel general game player. *KI*, 25(1):43–47, 2011.

[118] Kamil Rocki and Reiji Suda. Massively parallel monte carlo tree search. In *Proc. 9th Int'l Meeting High Performance Computing for Computational Science*, 2010.

[119] Yusuke Soejima, Akihiro Kishimoto, and Osamu Watanabe. Evaluating root parallelization in Go. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):278–287, 2010.

[120] Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Thomas Hérault, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Paul Vayssière, and Ziqin Yut. Scalability and parallelization of monte-carlo tree search. In *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers*, pages 48–58, 2010.

[121] Alan Fern and Paul Lewis. Ensemble monte-carlo planning: An empirical study. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011.* AAAI, 2011.

[122] Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto, and Yutaka Ishikawa. Scalable distributed monte-carlo tree search. In Daniel Borrajo, Maxim Likhachev, and Carlos Linares López, editors, *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011.* AAAI Press, 2011.

[123] Francois van Niekerk, Gert-Jan van Rooyen, Steve Kroon, and Cornelia P. Inggs. Monte-carlo tree search parallelisation for computer go. In Jan H. Kroeze and Ruth de Villiers, editors, *2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012*, pages 129–138. ACM, 2012.

[124] Junji Nishino and Tetsuro Nishino. Parallel monte carlo search for imperfect information game daihinmin. In *Fifth International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2012, Taipei, Taiwan, December 17-20, 2012*, pages 3–6. IEEE, 2012.

[125] Mehdi Goli, John McCall, Christopher Brown, Vladimir Janjic, and Kevin Hammond. Mapping parallel programs to heterogeneous CPU/GPU architectures using a monte carlo tree search. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2013, Cancun, Mexico, June 20-23, 2013*, pages 2932–2939. IEEE, 2013.

[126] Albert Xin Jiang, Leandro Soriano Marcolino, Ariel D. Procaccia, Tuomas Sandholm, Nisarg Shah, and Milind Tambe. Diverse randomized agents vote to win. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2573–2581, 2014.

[127] Leandro Soriano Marcolino, Albert Xin Jiang, and Milind Tambe. Multi-agent team formation: Diversity beats strength? In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. IJCAI/AAAI, 2013.

[128] Leandro Soriano Marcolino and Hitoshi Matsubara. Multi-agent Monte Carlo Go. In *10th Int'l Conf, on Autonomous Agents and Multiagent Systems*, AAMAS '11, pages 21–28, 2011.

[129] Nick Sephton, Peter I. Cowling, Edward J. Powley, Daniel Whitehouse, and Nicholas H. Slaven. Parallelization of information set monte carlo tree search. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014*, pages 2290–2297. IEEE, 2014.

[130] Lars Schaefers and Marco Platzner. Distributed Monte Carlo Tree Search: A novel technique and its application to computer Go. *IEEE Trans. Comput. Intellig. and AI in Games*, 7(4):361–374, 2015.

[131] Fatemeh Golpayegani, Ivana Dusparic, and Siobhán Clarke. Collaborative, parallel monte carlo tree search for autonomous electricity demand management. In *2015 Sustainable Internet and ICT for Sustainability, SustainIT 2015, Madrid, Spain, April 14-15, 2015*, pages 1–8. IEEE, 2015.

[132] Sayyed Ali Mirsoleimani, Aske Plaat, H. Jaap van den Herik, and Jos Vermaseren. Parallel monte carlo tree search from multi-core to many-core processors. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3*, pages 77–83. IEEE, 2015.

[133] Claude E Shannon. Computers and automata. *Proceedings of the Institute of Radio Engineers*, 41(10):1234–1241, 1953.

[134] Vadim V. Anshelevich. The game of hex: An automatic theorem proving approach to game programming. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, pages 189–194. AAAI Press / The MIT Press, 2000.

[135] Vadim V Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134(1):101–120, 2002.

[136] Jack van Rijswijck. Search and evaluation in hex. *Master of science, University of Alberta*, 2002.

[137] Jing Yang, Simon X. Liao, and Miroslaw Pawlak. New winning and losing positions for 7x7 hex. In Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson, editors, *Computers and Games, Third International Conference, CG 2002, Edmonton, Canada, July 25-27, 2002, Revised Papers*, volume 2883 of *Lecture Notes in Computer Science*, pages 230–248. Springer, 2002.

[138] Ryan Hayward, Yngvi Björnsson, Michael Johanson, Morgan Kan, Nathan Po, and Jack van Rijswijck. Solving 7x7 hex with domination, fill-in, and virtual connections. *Theor. Comput. Sci.*, 349(2):123–139, 2005.

[139] Gábor Melis and Ryan Hayward. Six wins hex tournament, 2003.

[140] Broderick Arneson, Ryan Hayward, and Philip Henderson. Wolve wins hex tournament. *ICGA Journal*, 32(1):49–53, 2008.

[141] Rune Rasmussen and Frédéric Maire. An extension of the h-search algorithm for artificial hex players. In Geoffrey I. Webb and Xinghuo Yu, editors, *AI 2004: Advances in Artificial Intelligence, 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, December 4-6, 2004, Proceedings*, volume 3339 of *Lecture Notes in Computer Science*, pages 646–657. Springer, 2004.

[142] Philip Henderson and Ryan B. Hayward. Probing the 4-3-2 edge template in hex. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008*, volume 5131 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2008.

[143] Fabien Teytaud and Olivier Teytaud. On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms. In Georgios N. Yannakakis and Julian Togelius, editors, *Proc. 2010 IEEE Conference on Computational Intelligence and Games, CIG 2010, Copenhagen, Denmark, 18-21 August, 2010*, pages 359–364. IEEE, 2010.

[144] Ryan B Hayward and Jack Van Rijswijck. Hex and combinatorics. *Discrete Mathematics*, 306(19):2515–2528, 2006.

[145] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte carlo tree search in hex. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):251–258, 2010.

[146] Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. Mohex 2.0: A pattern-based MCTS hex player. In *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013. Revised Selected Papers*, pages 60–71. Springer, 2013.

[147] Ikuro Ishigure. *In the Beginning.* Ishi Press, 1973.

[148] George Karypis. CLUTO a Clustering Toolkit. http://www.cs.umn.edu/ karypis/cluto/, October 2006.

[149] Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *Proc. 11th Int'l Conference on Information and Knowledge Management*, pages 515–524. ACM, 2002.

[150] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego - an open-source framework for board games and Go engine based on Monte Carlo Tree Search. *Computational Intelligence and AI in Games, IEEE Trans. on*, 2(4):259–270, Dec 2010.

[151] Peter Drake. Orego. https://sites.google.com/a/lclark.edu/drake/research/orego, 2012.

[152] Markus Enzenberger. Gogui. http://gogui.sourceforge.net, 2012.

[153] Richard B. Segal. On the scalability of parallel UCT. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers*, volume 6515 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2010.

[154] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Benzene. http://benzene.sourceforge.net, 2012.