Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 14-019

Improving Play of Monte-Carlo Engines in the Game of Go

Erik Steinmetz, Maria Gini

August 12, 2014

Improving Play of Monte-Carlo Engines in the Game of Go

Erik Steinmetz Maria Gini
University of Minnesota
{steinmet|gini}@cs.umn.edu

Abstract

We explore the effects of using a system similar to an opening book to improve the capabilities of computer Go software based on Monte Carlo Tree Search methods. This system operates by matching the board against clusters of board configurations from games played by experts. It does not require an exact match of the current board to be present in the expert games.

Experimentation included results from over 120,000 games in tournaments using the open source Go engines Fuego, Orego, Pachi, and Gnugo. The parameters of operating our matching system were explored in over thirty different combinations to find the best results.

We find that this system through its filtering or biasing the choice of a next move to a small subset of possible moves can improve play even though this can only be applied effectively to the initial moves of a game.

1 Introduction

In the last decade a new approach to playing games which are not amenable to traditional tree search algorithms has arisen. This approach uses a stochastic method to evaluate nodes in the game tree, and is often called Monte Carlo search. The principle of stochastic evaluation is to score a node in the game search tree, not by using a heuristic evaluation function, but by playing a large number of test games using randomly chosen moves starting at the node to be scored out to the end of the game.

In this paper we discuss experimental testing of 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 at the beginning of the game which have not been played by professional Go players in similar situations. Because of the difficulty in creating reasonable opening books for Go, due to the breadth of play, we create a score 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 very 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.

2 Description of the Game

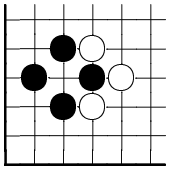
The ancient game of Go, called Baduk in Korean and Weiqi in Chinese, is a two-player, perfect information game played by placing black or white stones on a 19 by 19 grid of lines, with the black and white players alternating turns. Once placed, the stones are not moved around on the board as are chess pieces, 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.

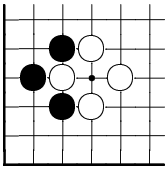
The object of the game is to surround or control the territory on the board. This is done by walling off sections of the board with stones of one's own color such that the opponent cannot create a living group of stones within your controlled area. A stone or group of stones is alive, or avoiding capture, 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.

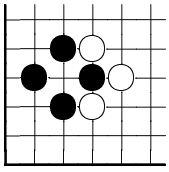
It is possible to capture enemy stones by denying them access to an open intersection, also called an eye point. For example, a single stone can be captured by placing four enemy stones on each of the adjacent intersections. Any number of stones may be captured at a single time if the enemy group has no remaining eyes.

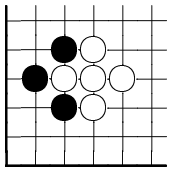
Only two rules restrict the placement of a stone. The first rule is a no-suicide rule: a player may not place a stone that would cause the immediate capture of the player's stone or group of stones. Since one gauges the effects on the opponent's stones first, however, there are many moves which capture an opponent's group which temporarily would violate this rule.

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 called a 'ko' as shown in the following diagram.


Beginning of Ko


White Captures


Black Recaptures


White Fills the Ko

The 'ko' rule prevents the third move in the sequence above, 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 above, or replying to black's move and allowing black to retake the ko.

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 (Masayoshi, 2005).

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.

At all skill levels of play, 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. Additionally, professional players are ranked on a separate scale, such that a top-ranked amateur would be equivalent to a 1 dan professional. Professional dan ranking ranges from 1 dan to 9 dan.

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. Typically an extra stone is given for each level of difference in the amateur 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. The strength differences in professional dan rankings are much smaller, and so professionals always play even games. To offset the first-move advantage in an even (non-handicap) game, a 'komi' of 4.5 points or 7.5 points is granted to white. For a 4.5 point komi, black must win by five points in order to secure a 1/2 point victory.

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 being 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 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 $200 * \sqrt{2}$. 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 100 Elo points.

3 Prior Work on Computer Go

There have 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, with many programs still competing to claim top honors. Through the eighties and nineties development of Go software proceeded slowly using variations of traditional techniques and reached approximately the 10 kyu level of play: that of a medium-strength beginner.

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 (Richards and Hart, 1961) and (Brudno, 1963)). 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.

Because the expansion factor of the game tree is much larger than that of chess, and more importantly because of the difficulty in creating any reasonable evaluation function, most Go-playing programs used an architecture first used in (Zobrist, 1970) that depended on having different code modules supply suggested moves and related urgency scores to the main program, which then chose among this limited set of moves.

In the last decade, much effort has been spent on applying Monte Carlo techniques to Go. This was first proposed by Brügmann in 1993 (Brügmann, 1993). It completes games with random moves many thousands of time from a given position, and scores a move choice based on the outcome of this random sampling of games. Although initial results were poor, these efforts progressed rapidly to the point that a program named MoGo utilizing Monte Carlo methods was able to defeat a professional player on a 19×19 board, albeit with a large handicap advantage.

3.1 The architecture of Monte Carlo and UCT Go programs

Monte Carlo methods to evaluate game positions are based on playing a large number of sample games to completion beginning from the position or move to be evaluated, choosing random legal moves until the end of the game, where a winner for that playout is determined. In an unbiased Monte Carlo search for Go, the moves chosen in these playouts comply to the rules of no suicide, and no playing into a ko, but otherwise are selected at random from the open intersections on the board. The candidate moves are then scored by counting the number of times the random games resulted in a win vs. the number of times they resulted in a loss. Another method of scoring a candidate move involves not just the number of random games won and lost, but also the amount (number of stones) by which these games were won or lost (Yoshimoto et al., 2006).

In its simplest form, also known as flat Monte Carlo, all possible moves from the current position are evaluated with an equal number of playouts. For example, as shown in Figure 1 if there were eighty possible moves (ignoring symmetry) available on a 9×9 Go board for the next play, nodes would be created for all eighty of those positions, and then random playouts would be conducted beginning at each of those positions, using an equal number of playouts to create a score for each node. In this example if 1000 playouts were conducted from each of the nodes for a total of 80,000 playouts, the node containing the highest number of wins would be chosen as the next move.

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 increase, due to the lack of an opponent model (Browne, 2011).

In Monte Carlo Tree Search (MCTS), see Figure 2, a game tree is constructed with the current situation as the root node. A single node is added to the tree at a time, and a randomized playout beginning from the newly added node is run. When the playout is complete, statistics in the new node and all of its parents are updated with the result of the playout: when the result for a playout is determined, each node that led to that result has its score modified by the win or loss. The algorithm and formula used to choose which node to add to the tree is called the 'in-tree' policy of the MCTS go engine. The algorithm that chooses the moves in the random playout to the end of the game, the out-of-tree

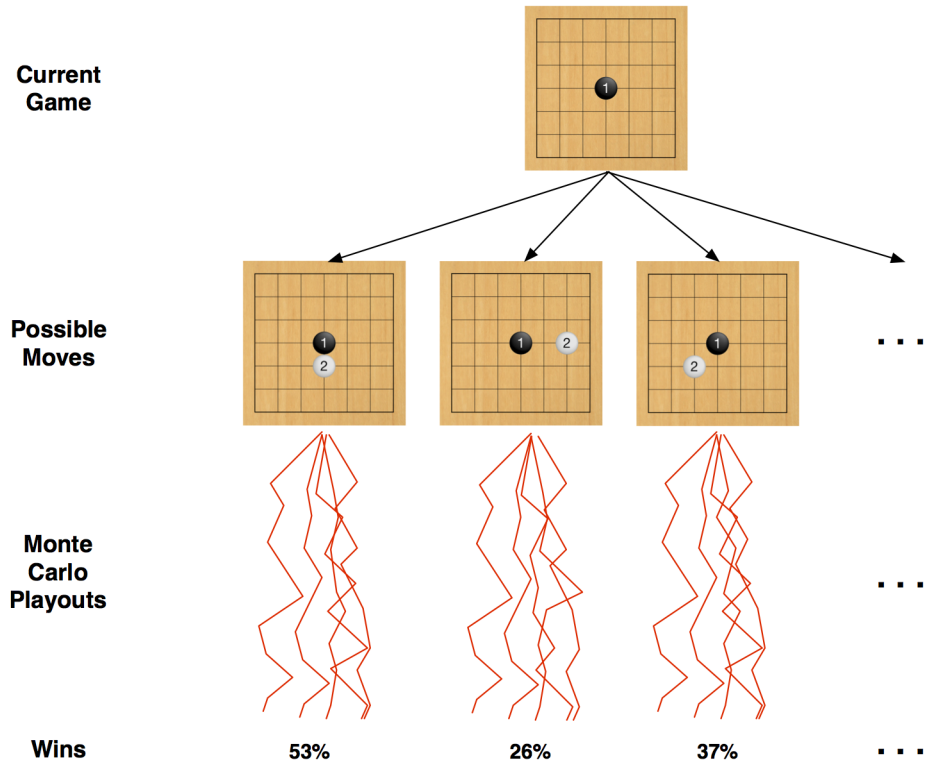


Figure 1: Possible second moves in a 9×9 game, each evaluated using Monte Carlo playouts. The move which has the greatest winning percentage is chosen as the best option.

policy, is often called the default policy.

In order to make decisions about which game tree nodes to traverse or expand, each node in the tree must at least keep track of the number of times it has been played (chosen), the number of times each following move was played, and the number of recorded wins from each next move. These statistics are then used to make the in-tree choice of next move, traversing it if the node already exists in the tree, or adding it to the tree if not already present. An example of adding nodes to the tree is shown in Figure 3. During Playout 1 in the example, Node B (representing a particular move following the game situation represented by Node A) is added to the tree. It is chosen as the successor to Node A using the in-tree selection policy. Then starting from the Node B position, a game is played out to the end with random but legal moves. At the end of the game it is determined that this random playout has resulted in a victory, so the win is assigned to node B which will then have one win out of one try. The win is also propagated up the tree to Node A, which will increment both its win counter and its try counter. During Playout 2 Node B is chosen again using the in-tree policy, and then the Node C is created and added to the tree. A playout beginning at the Node C position results in a loss so C is assigned zero wins with one try, while both Nodes B and A also have their try counters incremented. In Playout 3 Node D is created as the successor to Node A, and a playout from D results in a win, assigning one win from one try to node D, and incrementing node A's win and try counters.

The selection policy of an in-tree node is different than the selection policy of an out-of-tree node. The out-of-tree move selection policy, called the default policy, is often a completely random selection, limited only by the legality of the move. The in-tree move selection policy in basic MCTS is to choose the move with the best-so-far win rate, thereby causing the more favorable parts of the tree to be examined in more depth than others. Usually the win rate of an unplayed move in a tree node is assigned infinity so that all unplayed options from a particular node are tried at least once.

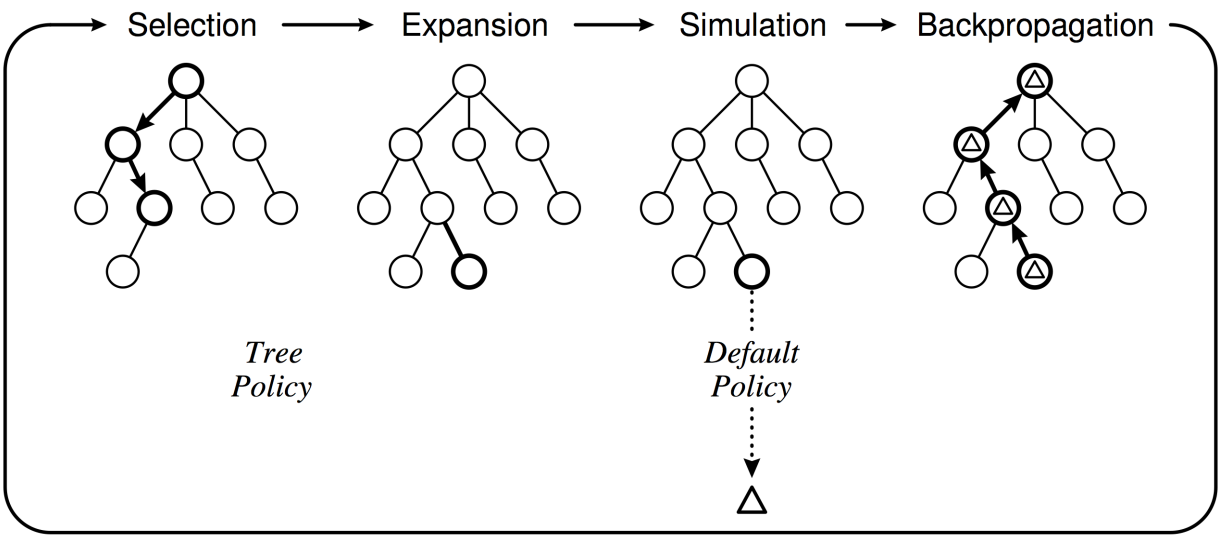


Figure 2: Monte Carlo Tree Search Process.

Because each move in a test play-out is generated randomly and checked only for its legality, or picked from available choices with a very simply policy, a game can be played to a conclusion very rapidly. Due to the speed with which these games can be played to the end, it is possible to run many thousands of random games for each candidate move in a reasonable amount of time even on a single processor machine. For example, results show that one hundred thousand playouts on a 2.8 GHz machine can be accomplished in approximately three seconds (Cook, 2007).

A modification to the MCTS algorithm that is now widely used has an extra variable to assist the selection of the in-tree moves (Kocsis and Szepesvari, 2006). This method, called Upper Confidence Bounds Applied to Trees (UCT), chooses moves in the tree by iterating through the scoring of candidate moves as with the normal Monte Carlo algorithm but uses the number of times successor nodes in the tree are encountered along with the their win rate as a modification to the basic in-tree selection policy. If a move has only been tried a few times, its winning rate will have a low confidence, and so in order to increase the confidence of the winning rate, the odds of selecting these moves are increased by an additional factor so that they will be explored more often.

A additional modification, called RAVE for rapid action value estimation, scores moves not just based on playing at the current position (of the node being scored) but using a formula based on playing that move at any point during the game. The formulation is known as the all-moves-as-first heuristic (Gelly and Silver, 2011). Using this method, if the overlap of successor nodes in the tree is small, the scoring will degenerate into normal Monte Carlo evaluation, but if there is enough overlap, the concentration of the work on the most promising nodes should lead to better results. One of the best programs currently playing, MoGo, was built upon this basis.

Other modifications include integrating domain dependent knowledge (Bouzy, 2007) and heuristic (Drake and Uurtamo, 2007) methods along with a system for polling different Go programs for move choices (Marcolino and Matsubara, 2011).

One interesting modification involves a simple heuristic in the default playout policy which stores for each player the most recent winning reply move to a given move (Drake, 2009). Thus in the random 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 which led to a win for the currently moving player. Because this involves only 361 possibilities for each player, the lookup table is very 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 (Baier and Drake, 2010).

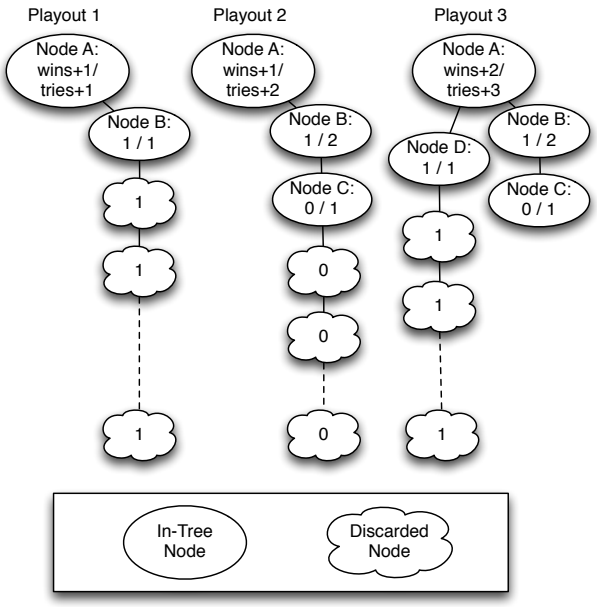


Figure 3: Three playouts starting at Node A. In each playout, the first out-of-tree node is added to the tree. All in-tree nodes of the playout have their tries and wins counters updated.

3.2 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 very 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 (Ishigure, 1973), 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.

Limited opening books exist in many computer-Go programs, but most of the included sequences in the books are quite short, including only four or five moves. Some longer sequences, though included in the books, are rarely invoked.

4 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.

4.1 Fuego

Fuego (Enzenberger et al., 2010) 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 very well in computer to computer Go tournaments, with an occasional top place, and often scoring in the top three.

4.2 Orego

Orego (Drake, 2012) 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.

4.3 Pachi

Pachi (Baudiš and Gailly, 2011) 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.

4.4 Gnugo

Gnugo (Bump and Farnebäck, 2008) 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 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.

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 4 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.

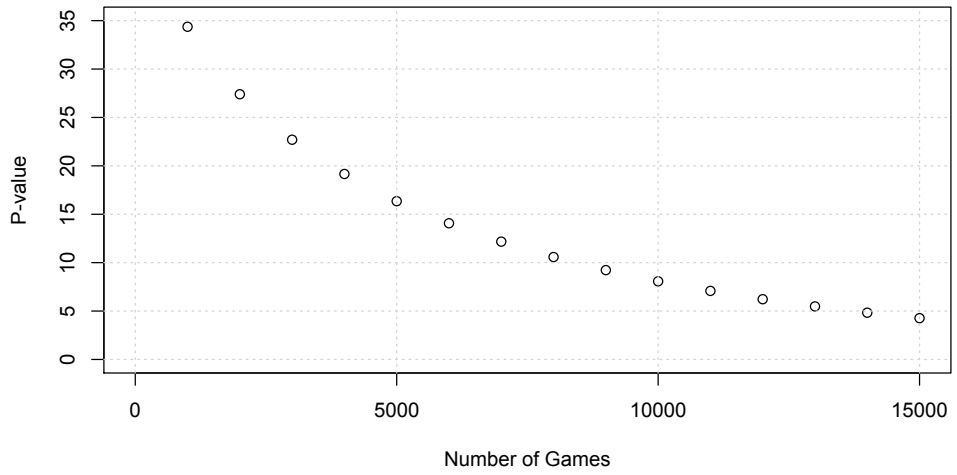


Figure 4: The p-values calculated for comparing a 51% winning rate versus a 50% winning rate.

6 Current Go Servers

There are a number of public and private go servers around the world which allow people to play go with each other, with go engines, and some which allow go engines to play with other go engines. Playing a go game on a server, for a human, usually involves operating some client software on a desktop PC with a graphical user interface that connects to the server, displaying the current state of the board on a screen, updating it with opponent and player moves, and transmitting the players moves to the server.

The most popular public servers are the Internet Go Server also known as PandaNet, KGS Go Server, Tygem-Baduk, and WBaduk. These are mostly used for people to play other people, but go engines are allowed to play, and many people will play against them. There is also a popular computer-to-computer specific go server called CGOS (Computer GO Server). This server allows go engines to compete with each other almost continuously, or in one of many periodic, online tournaments. One of these is called the computer go ladder, which is an ongoing 'tournament' allowing computer go engines to play against each other for the purpose of being ranked.

7 Computer vs. Human Play

There is now a reasonable amount of human vs. computer play on most of the go servers, but games between top ranked professional players and computers are still quite rare. An annual tournament exists, called the UEC Cup Computer Go where computer go engines compete in a tournament in order to gain the privilege of playing against a professional player. When a professional go player has a match against a computer, it is typically run in a fashion similar to a face-to-face game. A physical board is used to play the game, and the computer is represented by a human in the room who places stones for the computer on the board, and enters the professional's moves into the computer.

Play between humans and computers on the popular go servers consist mostly of games between amateur players and the go engines. The go engines are able to be ranked similarly to human players using either the kyu and dan system, or with an Elo rating.

8 Using Professional Play in the Opening Game to Improve Monte Carlo Search

One of the goals of the current research is to show how the play of Monte Carlo-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 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 moves 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. Similar to an opening book we assume that in the beginning of the game all moves played by professional players are some of the best moves known, regardless of the final outcome.

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

During a Monte Carlo search, all legal moves are considered by the search engine 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, 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 score them.

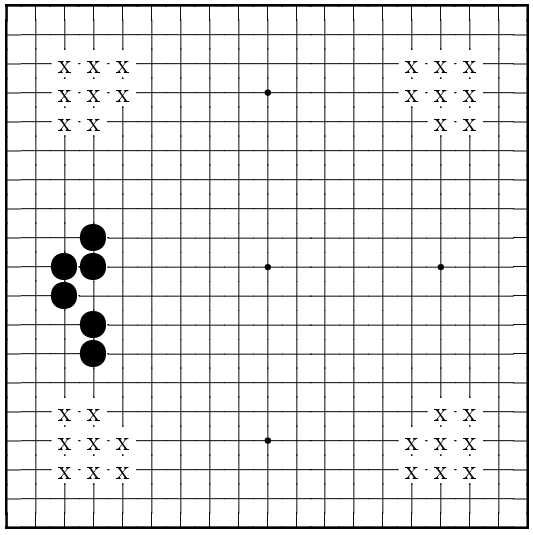


Figure 5: Unusual Opening Moves in Black. Typical Openings Marked by 'x'.

Because so many different moves are being considered that are not at all viable, the Monte Carlo 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. For example, Figure 5 shows some opening moves chosen by a UCT program which would never be chosen by a professional player, or even an amateur (shown here in only one quarter of the board, these moves were repeated on each side of the board).

Our work involves pruning the search by only considering moves that have been made by professional players in the same or similar situations. This approach differs 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.

In a normal Monte Carlo or UCT search from a given board position all legal moves on the board are considered at least a small number of times before the bias towards already successful trials abbreviates the searches in some of

those options. If a 'bad' move, one that is recognizably bad to even an amateur player, gets some luck on its first trials, it is possible that the search will spend a large amount of time exploring the sub-tree from that move. By eliminating these sub-trees from the search, we allow the program to spend more of its resources on exploring moves that are considered 'good' by human players. The scale of this reduction at the beginning of the game is substantial. For example, in Figure 6 we consider a game after 8 moves. The program is searching for the next black move. With a normal search, it would start sub-trees at all open intersections on the board, $(19 \times 19 - 8)$, which is 353 different possibilities. By only considering the moves played by professionals in similar situations, we reduce the number of next-move possibilities to only 34 options, a ten-fold reduction:

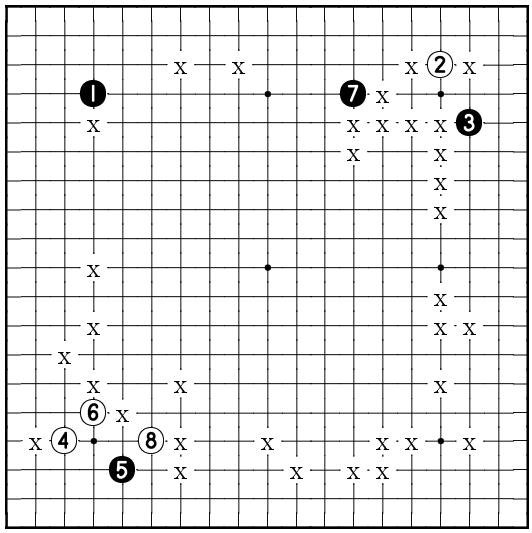


Figure 6: Example Opening to Move 8. Professional Moves for 9 Marked by 'x'.

To find a match for a board, we consider each of the points on the board to be two elements, giving each board situation a score in 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 element one would represent a white stone in the upper left corner of the board, while element two would represent a black stone in the same location. 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.

In order to find similar game positions in games played by professional human players, we have scored professional games from a large database and then clustered these together based on their similarity, as described later in Section 9.2.

9 Computer vs. Computer Experiments

In order to show improved performance of a Go-playing engine, we incorporated our SMARTSTART technique into the Monte Carlo Tree Search style "Orego" and "Fuego" programs. We then compared the results of these programs vs. the Gnugo program to the results of SMARTSTART versions of these programs playing against Gnugo. For each set of results we wished to run many thousands of games of the normal versions of these programs alongside the SMARTSTART versions of these programs.

Conducting the experiment consisted of three parts: creating the database of entries based on professional games, clustering the resulting entries so that an observed game can be matched against a set of similar professional games

rather than looking for an exact opening match, and playing the opening moves of games drawing from next-moves available in a matched cluster.

9.1 Database Creation

A large database of full 19 $\times$ 19 board professional level Go games was used as the basis for finding patterns in opening play. Each game produced 160 entries in the database, eight for each move in the game up through move twenty. Eight entries are needed for each board position due to the eight-way symmetry of the Go board. Any given position can be rotated through the four sides, and additionally mirrored over an axis. Each of these entries contains the position on the board, and the move that was played by the professional in response to that position. Since each game is an evenly-matched game, not a handicap game, move one will always be a black move, move two a white move, and so on.

Each entry in the database through move 12 was created as a 722 element vector of ones or zeroes. The set of 722 elements were created with two elements for each of the 361 points on the board. The two elements represent the presence or absence of white and black stones on a particular intersection.

In order to best facilitate quick and accurate matching, the database has been divided into twenty separate databases: one 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 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, each with seven stones on the board. The advantage of a divided database is that we can be assured that only the most similar board positions are being searched.

9.2 Clustering of Database Games

The entries from the professional games were clustered into 64, 96, 128, ... or 256 groups based on their similarity to one another as measured by their cosine distance in a 722 dimensional space where one considers each element in the vector describing the board as a single dimension. The clustering was accomplished with the Cluto program (Karypis, 2006) resulting in an entry for each cluster consisting of the dimensional score of the cluster's center, along with a list of the next moves played during the games in that cluster, and the frequency of those next moves.

When finding the closest match to a given board position, the position is expressed as a vector of elements, each a zero or a one and considered one of the dimensions. We calculate the similarity of board positions by using the cosine of all the dimensions. Thus two positions are similar if the vectors formed by their values in the multi-dimensional space point in the same direction. That is, they have values in similar proportions along all of their axes. The cosine measure was used both for the discovery of clusters in the database, and when matching a game with the discovered clusters.

9.3 Finding a move during play

With the database of professional games scored along the chosen dimensions (patterns), and grouped into clusters, we matched the positions of a game in progress against the cluster centers in the database in order to supply the possible next moves available to the Monte Carlo algorithm. At each point during a search where the agent was searching through the initial moves of the game we limited the options of the algorithm to those moves that had been played by professional players in the games contained in the nearest cluster to the current position. This was done both when the choice was inside the tree for node selection and when the choice was outside the tree for the default policy.

At each position under consideration, its multi-element score was created and then compared to the scores of each of the cluster centers for that move number. The next moves from the games contained in the nearest cluster were then offered as the only options for the Monte Carlo algorithm.

10 Tournament Setup

During the course of the summer research term and into the fall and winter of 2013, this project was able to utilize a 40-core NAIST computer[1] using the Windows Server operating system to run a very large number of games between various Go engines utilizing different parameters of the SMARTSTART algorithm. By testing out this technique applied in different Go engines and under varying conditions, we were able to measure the algorithm's performance with statistically significant results.

10.1 Applying SMARTSTART to Orego

Testing our implementation of SMARTSTART with the Orego program involved extending one of Orego's 'Player' classes, which control the in-tree node choices and overall play along with adding a 'Heuristic' class to add rules for the out of tree playout move decisions. Both of these modifications were for move choices through move 12 or 20 depending on the experiment, and filtered the available moves to only those that were chosen by professionals in the closest cluster of games in the database. These extra classes were then invoked as the player and heuristic classes respectively, and the Orego architecture was able to load them without significant modification of the original code.

The versions of Orego, with and without SMARTSTART, were matched up against Gnugo 3.8 set at skill level 10. Both versions of Orego, with SMARTSTART and unmodified, played with the number of playouts per turn fixed at 8,000. The version of Orego with SMARTSTART was played with different parameters, including the move number through which the SMARTSTART was invoked (either move 12 or move 20) and the number of clusters created in the pattern matching set. The number of clusters that were attempted were 64, 96, 128, 160, 192, 224, and 256. Finally, the size of the database sample[2] (the number of professional games therein) was varied with sizes of either 1000 games which we called small or 8000 games which we called medium.

10.2 Applying SMARTSTART to Fuego

Since the Fuego program is a much 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 Satsugen.

The application of the SMARTSTART technique to the Fuego program involved adding a class which implemented functions to provide lists of responses from the professional database given a position on the board during the first twelve moves. These tournaments utilized the smaller database sample of about 1000 games, with each move's games divided into 64 clusters.

In the first tournament these 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. Fuego played with the number of playouts set to 16,000.

In the second tournament, the matching responses were used to bias in-tree nodes. From a given position represented by a tree node, the follow- on board positions which had been chosen by professionals were given a bias of either 20 victories out of 20 games played, or 40 victories out of 40 games played. This bias causes the Monte-Carlo algorithm to favor exploring those nodes over those without the bias.

10.3 Fuego with Tripled Playout Limits

In order to explore the effects of extra computational effort at the beginning of the game, and compare it to the effects of our SMARTSTART modifications, a version of Fuego was created which used triple the number of playouts during the first twelve moves. Instead of invoking an opening book or utilizing the SMARTSTART filtering or biases, this version was allowed to use up to 48k playouts to build its game tree to choose the opening moves of the game.

This version of Fuego was run in a "self-play" tournament against an unmodified version of Fuego. Both versions were run with Fuego's opening book disabled.

[1]The Nara Institute of Science and Technology, Graduate School of Information Science, Software Engineering Lab led by Prof. Kenichi Matsumoto provided access to this server. NAIST is located in Ikoma City, Nara Prefecture, Japan.

[2]The database of games from which the samples were taken is the commercially available Games of Go on Disk (GoGoD) collection. These samples were taken from the Winter 2008 collection.

11 Tournament Simulations Results

11.1 Orego vs. Gnugo Results

In our tournaments of Orego, with and without SMARTSTART against Gnugo 3.8 we played 2000 games in each of the 28 different configurations outlined above to see if any sizable difference in playing abilities could be detected. With 2000 games per configuration, the 95% confidence interval for each is about 4.3% (p $\pm$ 2.16%).

At the same time that these were being run, the unmodified version of Orego (v 7.15) was also run against Gnugo. A total of 24,000 games were run in this configuration giving a 95% confidence interval of about 1.36% (p $\pm$ 0.68%).

The win rates of the various configurations of Orego and Orego with SMARTSTART are shown in Tables 1 and 2 and summarized in Figure 7. Most of the configurations ended up with similar or poorer winning ratios than normal Orego, with even the best results failing to show a statistically significant improvement.

Table 1: SMARTSTART Orego vs Gnugo, Small Database

Matching through Move 12			
	As Black	As White	Cumulative
Normal Orego	44.1%	40.1%	42.1%
SMARTSTART			
64 Clusters	42.9%	40.2%	41.55%
96 Clusters	44.1%	36.1%	40.1%
128 Clusters	45.5%	40.8%	43.15%
160 Clusters	44.3%	38.3%	41.3%
192 Clusters	43.6%	40.8%	42.2%
224 Clusters	42.9%	42.2%	42.55%
256 Clusters	42.6%	40.0%	41.3%

Matching through Move 20			
	As Black	As White	Cumulative
Normal Orego	44.1%	40.1%	42.1%
SMARTSTART			
64 Clusters	41.8%	39.6%	40.7%
96 Clusters	41.4%	39.8%	40.6%
128 Clusters	41.4%	39.8%	40.6%
160 Clusters	40.5%	36.8%	38.65%
192 Clusters	41.7%	41.5%	41.6%
224 Clusters	42.4%	39.2%	40.8%
256 Clusters	40.9%	37.5%	39.2%

Table 2: SMARTSTART Orego vs. Gnugo, Medium Database

Matching through Move 12			
	As Black	As White	Cumulative
Normal Orego	44.1%	40.1%	42.1%
SMARTSTART			
64 Clusters	42.4%	39.2%	40.8%
96 Clusters	44.1%	39.6%	41.85%
128 Clusters	41.2%	36.9%	39.05%
160 Clusters	44.6%	41.0%	42.8%
192 Clusters	42.5%	36.8%	39.65%
224 Clusters	44.0%	38.4%	41.2%
256 Clusters	42.2%	40.0%	41.1%

Matching through Move 20			
	As Black	As White	Cumulative
Normal Orego	44.1%	40.1%	42.1%
SMARTSTART			
64 Clusters	43.2%	38.7%	40.95%
96 Clusters	40.6%	36.4%	38.5%
128 Clusters	40.3%	38.1%	39.2%
160 Clusters	45.3%	40.0%	42.65%
192 Clusters	42.6%	39.0%	40.8%
224 Clusters	41.9%	40.4%	41.15%
256 Clusters	44.6%	37.5%	41.05%

11.2 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. Additionally 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.

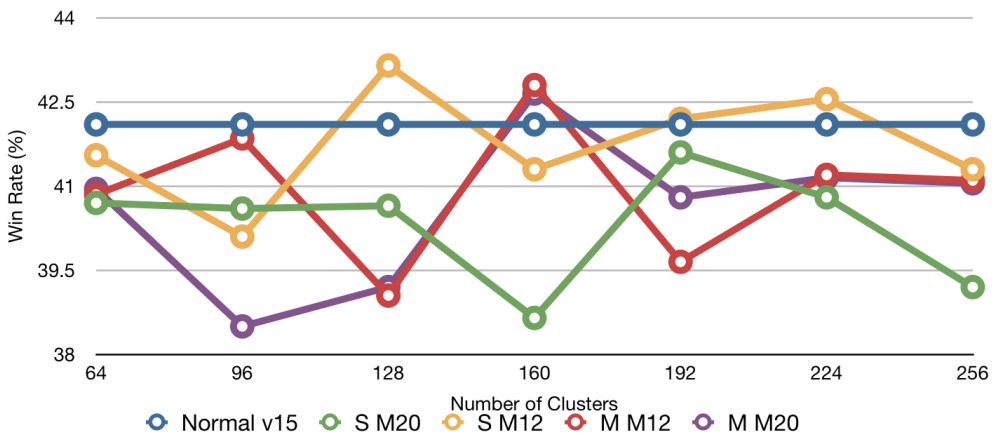


Figure 7: Summary of SMARTSTART Orego and Orego vs. Gnugo level 10. SMARTSTART configurations are indicated with S or M for small or medium database, and M12 or M20 for utilizing the SMARTSTART modification through move 12 or move 20.

The 14,000 game tournaments provide a 95% confidence interval of approximately 1.65% (p $\pm$ 0.825%). The win rates of these different configurations are shown in the following chart along with the p-value for the null hypothesis that the SMARTSTART Fuego version is equal or weaker than the unmodified Fuego. Additionally, a version of SMARTSTART Fuego was run with a 20 win bias in place of filtering. This version produced very poor results, with a win ratio two percent poorer than the unmodified Fuego. We are still investigating the causes of this strange result.

Table 3: Fuego vs. Pachi 10, Small Database

	As Black	As White	Cumulative	p value
Unmodified Fuego with no Opening Book	50.88%	57.66%	54.275%	-
Fuego with SMARTSTART Filtering Through Move 12	54.16%	56.54%	55.35%	4.24%
Fuego with SMARTSTART In-Tree Bias 40 Through Move 12	53.97%	57.71%	55.84%	0.58%

11.3 FuegoTriple vs. Fuego Results

The tournament of Fuego with triple the effort in the opening game vs. the unmodified Fuego consisted of 16,000 games, providing a 95% confidence interval of about 1.55% (p $\pm$ 0.777%).

Table 4: FuegoTriple vs. Fuego, Small Database

	As Black	As White	Cumulative
Fuego with Triple Playouts Through Move 12	51.21%	48.6%	49.59%

12 Computer to Human Server Results

This project worked on setting up the Koyama server to allow play between a select group of very strong amateur players and four different Go engines. We were able to compile and connect a standard version of Orego along with a

SMARTSTART version of Orego. Additionally both a standard version of Fuego and a SMARTSTART version of Fuego were compiled and connected to the server. Fuego and Orego code was also modified to correctly respond to playing in handicap games.

A small number of very skilled amateur players, ranked 3 to 5 dan, were asked to play against the various versions of computer Go engines on the server, and give their opinions about the perceived play of these programs.

13 Conclusions and Future Work

Looking over the various results of the Orego tournaments we can see that in very few configurations did the application of our SMARTSTART technique perform better than the current default Orego engine. Due to the large number of configurations tested, though, each had such a large confidence interval that no results can be considered statistically significant. Despite this, however, there appeared to be a trend that the use of SMARTSTART through move twelve appeared to perform better than the use of SMARTSTART through move twenty with the decreased dimensional size.

The results of applying the SMARTSTART technique to Fuego appeared to show some modest improvements. When either applied as a filter, or as a very large bias, the moves chosen by SMARTSTART created slightly better win rates with results that are statistically significant.

The lack of any discernible difference in self-play between Fuego and a version modified to use triple the effort in the first twelve moves of the game shows that it is difficult to change the outcome of a long game when only modifying its very first moves. When the win rates of MCTS Go engines are gauged against the number of playouts per move applied over the entire game, a tripling of the number of playouts, especially at levels less than 100,000, increases the win rate substantially (Enzenberger et al., 2010).

If our method is really improving the play, even by a small amount, as indicated by our results, then we can consider it to be more effective than increasing the amount of work done by a factor of three, at least over a small number of moves.

Continuation of this work will include testing the effects of our SMARTSTART system when the number of playouts per move is larger through the entire game, applying the system to the Pachi go engine, and continuing to explore the application of smaller dimensional pattern matching in later moves.

14 Acknowledgements

We gratefully thank Professor Emeritus Koyama and Professor Matsumoto of the Software Engineering Lab at NAIST for hosting and support during the 2013 summer program, along with their continued advice and assistance. This work has profited greatly from their generous help and encouragement.

Access to computer time, network resources, and abundant technical support was provided by the Nara Institute of Science and Technology, Graduate School of Information Science.

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.

References

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 Transactions on*, 2(4):303–309, Dec 2010.

Petr Baudiš and Jean-loup Gailly. Pachi: State of the Art Open Source Go Program. In *Advances in Computer Games 13*, Nov 2011.

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, 2007.

Cameron Browne. The Dangers of Random Playouts. *ICGA Journal*, 34(1):25–26, 2011.

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

Bernd Brügmann. Monte Carlo Go. October 1993. URL `ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/mcgo.ps`.

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

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. URL `http://computer-go.org/pipermail/computer-go/2007-April/009640.html`.

Peter Drake. The Last-Good-Reply Policy for Monte-Carlo Go. *International Computer Games Association Journal*, 32(4):221–227, 2009.

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

Peter Drake and Steve Uurtamo. Heuristics in Monte Carlo Go. In *Proceedings of the 2007 International Conference on Artificial Intelligence*, 2007.

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 Transactions on*, 2 (4):259–270, Dec 2010.

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

Ikuro Ishigure. *In the Beginning*. Ishi Press, 1973.

George Karypis. Cluto a Clustering Toolkit. http://www.cs.umn.edu/ karypis/cluto/, October 2006. URL `http://www.cs.umn.edu/ karypis/cluto/`.

Levente Kocsis and Csaba Szepesvari. Bandit based Monte-Carlo planning. In *ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

Leandro Soriano Marcolino and Hitoshi Matsubara. Multi-agent Monte Carlo Go. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, pages 21–28. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

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

D. J. Richards and T. P. Hart. The Alpha-Beta Heuristic. AI Memos 30, December 1961.

Haruhiro Yoshimoto, Kazuki Yoshizoe, Tomoyuki Kaneko, Akihiro Kishimoto, and Kenjiro Taura. Monte Carlo Go has a way to go. In *Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, pages 1070–1075, 2006.

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