Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Mining Expert Play to Guide Monte Carlo Search in the Opening Moves of Go

Erik Steinmetz and **Maria Gini**
Department of Computer Science and Engineering
University of Minnesota
{steinmet|gini}@cs.umn.edu

Abstract

We propose a method to guide a Monte Carlo 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.

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, called Monte Carlo search, to evaluate nodes in a game tree. The principle of stochastic evaluation is to score a node in a 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 evaluated out to the end of the game.

In this paper we introduce 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. 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 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 Prior Work on Computer Go

Go 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. The object of the game is to surround or control the territory on the board. 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 their turn. The game ends when both players have passed in succession and the winner is the player who controls the most territory. There are two scoring systems, Chinese and Japanese, but they both produce equivalent results, with a possible one stone variance [Masayoshi, 2005].

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

Because the growth of the game tree is much larger than that of chess, and more importantly because of the difficulty in creating reasonable evaluation functions, most early Go 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 recent years, much effort has been spent on applying Monte Carlo techniques to Go [Browne *et al.*, 2012]. This was first proposed by Brügmann in 1993 [Brügmann, 1993], and completes games with random move sequences to the end of the game many thousands of time from a given position. A move choice is then evaluated based on the outcome of this random sampling of games. Although initial results were poor, the quality of this style of algorithm 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.

2.1 The architecture of Monte Carlo 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 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 losses. 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. 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 [Browne, 2011].

In Monte Carlo Tree Search (MCTS), moves of a playout are given nodes and added to a game tree. Due to memory constraints, only the first move played outside of the existing tree during a playout is given a node and added to the tree. When the result for a playout is determined, each node in the tree that led to that result has its score modified by the win or loss. Nodes keep track of the number of times played, the number of times each next move was played, and the number of recorded wins from each next move. When selecting a move during a playout, the selection policy from an in-tree node is usually different than the selection policy from an out-of-tree node. The out-of-tree move selection policy, called the default policy, is often a 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.

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 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.

An 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. This is known as the all-moves-as-first heuristic [Gelly and Silver, 2011]. One of the best programs currently playing, MoGo, was the first to be built upon this basis.

Other modifications include integrating domain dependent knowledge [Bouzy, 2007] and heuristic methods [Drake and Uurtamo, 2007] along with a system for polling different Go programs for move choices [Marcolino and Matsubara, 2011]. Biasing in-tree node selection based on learned patterns [Michalowski *et al.*, 2011] has improved play and modifying just the out-of-tree playouts includes using a lookup table based on individual move sequences from winning playouts [Drake, 2009] and was improved by removing from the table those sequences which were subsequently played in a losing playout [Baier and Drake, 2010].

2.2 Opening books in Computer Go

One of the ways in which computer chess programs have succeeded in reaching grand master levels is through learning the opening moves of a game from human experience. Chess masters discovered and developed over the years the most powerful sequences of opening moves, so one of the fastest ways for a student of chess to improve their game was to study these opening books and memorize 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 look up the best move for 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 running a search at the beginning of the game when the game tree is at its widest and longest.

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 with the depth and breadth of 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 section of the board, typically a corner. In [Baier and Winands, 2011] both full board matching and joseki (corner) matching are used to modify the MCTS search tree and playouts of a Go program, with some improvements to the winning rate of their platform program. This method required an exact match of either the entire board or of one of the corners to the selection of stored positions. When a match does occur, the next move from the matched game or games is not directly chosen, but instead used to either filter choices in an MCTS search or, more successfully, to bias the choices in an MCTS search tree.

3 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.

3.1 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 is quite small, and play leaves the book very 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. 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 was not able to provide a statistically significant improvement to play (see Table 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.

Table 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

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

In this paper 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 64 clusters, the average number of next moves played by professionals in each of the clusters after move 8 is about 34. By only considering these moves, we reduce the number of next move possibilities to only 34 options, a ten-fold reduction.

4 Design of SMARTSTART

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.

4.1 Database Creation

A sample of games spanning from 1980 to 2007 from a large database of full 19 $\times$ 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. Each professional game produced 8 entries in a database for each move in the game up through move twelve or twenty, for a total of 96 or 160 entries per professional game. Eight entries were used 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. The number of entries in the database was increased eightfold rather than rotating 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 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, for each move number, we took all the entries (8368 for the 1046 game sample) and clustered them into a much smaller (64 to 256) number of clusters.

4.2 Clustering of Database Games

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. A perfect match will be 1. 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. If the number of clusters is very low, 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 [Karypis, 2006]. 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 is large and the number of clusters is also large, these two methods end with very similar results, with a slight edge for partitioning [Zhao and Karypis, 2002]. Therefore we chose the top-down bisection method. For the similarity measure we used the commonly adopted cosine measure.

For each of the clusters created by Cluto, we created a single cluster entry in our comparison database to be used during play. These entries consist of the normalized vector representing the cluster's center, along with a list of the next moves played during the games that are included in that cluster, and the frequencies of those next moves.

We have looked at 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 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 be when the cluster contains, as one of its next moves, the move actually played in the observed sample 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 1. 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.

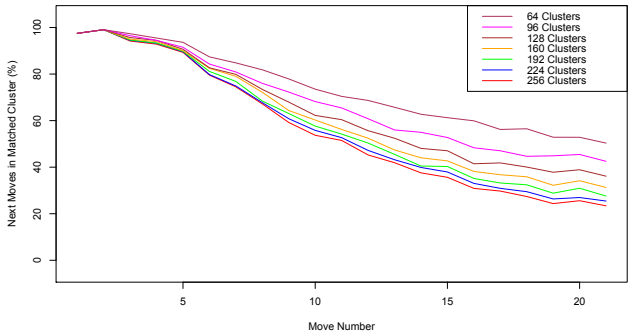

Figure 1: Percentage of next moves found in closest cluster.

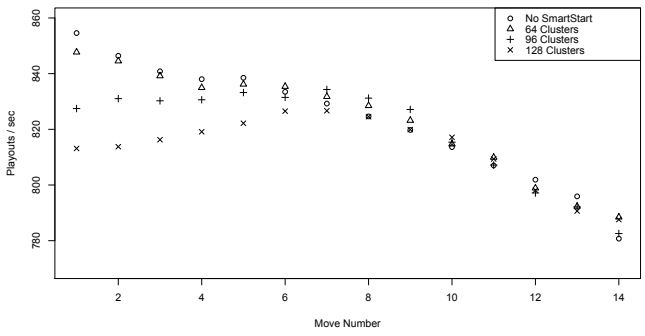

Figure 2: SMARTSTART Speed in playouts/s.

4.3 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 [Gelly and Silver, 2007] 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 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 2.

5 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 [Drake, 2012] program, written in Java. We played Orego against the older, non Monte Carlo Go program Gnugo [Bump and Farnebäck, 2008]. We also incorporated our method into the stronger Fuego [Enzenberger *et al.*, 2010] program, written in C++, and played it against another very strong open-source MCTS Go program called Pachi [Baudiš and Gailly, 2011].

5.1 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% ($p \pm 0.825\%$).

5.2 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 [Enzenberger, 2012]. 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 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.

5.3 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 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 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 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 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 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

6 Conclusions 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.

Acknowledgements

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. We 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.

References

[Baier and Drake, 2010] 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.

[Baier and Winands, 2011] 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.

[Baudiš and Gailly, 2011] Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source Go program. In *Advances in Computer Games 13*, Nov 2011.

[Bouzy, 2007] 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.

[Browne *et al.*, 2012] 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.

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

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

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

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

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

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

[Enzenberger *et al.*, 2010] 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.

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

[Gelly and Silver, 2007] 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.

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

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

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

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

[Marcolino and Matsubara, 2011] 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.

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

[Michalowski *et al.*, 2011] 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.

[Yoshimoto *et al.*, 2006] 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.

[Zhao and Karypis, 2002] Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 515–524. ACM, 2002.

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