Monte Carlo Go Has a Way to Go

Haruhiro Yoshimoto
Department of Information and Communication Engineering
University of Tokyo, Japan
hy@logos.ic.i.u-tokyo.ac.jp

Kazuki Yoshizoe
Graduate School of Information Science and Technology
University of Tokyo, Japan
yoshizoe@is.s.u-tokyo.ac.jp

Tomoyuki Kaneko
Department of Graphics and Computer Sciences
University of Tokyo, Japan
kaneko@graco.c.u-tokyo.ac.jp

Akihiro Kishimoto
Department of Media Architecture
Future University-Hakodate, Japan
kishi@fun.ac.jp

Kenjiro Taura
Department of Information and Communication Engineering, University of Tokyo, Japan
tau@logos.ic.i.u-tokyo.ac.jp

Abstract

Monte Carlo Go is a promising method to improve the performance of computer Go programs. This approach determines the next move to play based on many Monte Carlo samples.

This paper examines the relative advantages of additional samples and enhancements for Monte Carlo Go. By parallelizing Monte Carlo Go, we could increase sample sizes by two orders of magnitude. Experimental results obtained in 9×9 Go show strong evidence that there are trade-offs among these advantages and performance, indicating a way for Monte Carlo Go to go.

Introduction

Games have been extensively studied for over 50 years as testbeds for AI. A lot of successful techniques have been invented to improve the strength of game-playing programs.

It is widely known that incorporating efficient search algorithms is one of the most important factors for strong game-playing programs. Although there are some pathological cases (Nau 1980), Thompson (1982) showed that there is a strong correlation between the depth of the search trees explored by the programs and their playing strength. When comparing a program performing $(d+1)$-ply search against one doing d-ply search, many experiments in many games show that deeper search usually achieves improvements to the strength (e.g., (Billings & Björnsson 2003; Heinz 2001; Junghanns *et al.* 1997)). However, these experiments also confirm that diminishing returns for additional search eventually appears. The larger d becomes, the smaller difference in strength between $(d+1)$ and d-ply search is observed. This phenomenon suggests that other approaches must be investigated, when the time for additional search to achieve only small improvement comes.

Computer Go is a very challenging topic for game programmers. Because of a large branching factor and difficulties in position evaluation, Go remains resistant to current AI techniques, including a search-based approach. So far, despite a lot of effort, a player of medium skill can easily win against state-of-the-art computer Go programs (Müller 2002). Researchers therefore have been investigating new ideas to improve computer Go programs.

One of the most interesting and exciting ideas is a *sampling-based* approach, which is called *Monte Carlo Go* (Bouzy & Helmstetter 2003; Brügmann 1993). Monte Carlo Go collects a lot of samples to approximate the expected outcome for each move. It then selects the move that has the highest expected outcome. Each sample consists of a random game in which each player almost randomly selects a legal move until the final score of that random game is determined. Bouzy showed that Monte Carlo Go is very promising. In 9×9 Go, although his Monte Carlo Go program OLGA contains little Go-dependent knowledge, it performs better than his previous program INDIGO, which contains a search-based approach with a lot of hand-coded Go dependent knowledge (Bouzy 2003). Furthermore, Crazy Stone (Coulom 2005), which combined Monte Carlo Go with tree search, won the 10th KGS computer Go tournament in 2006.

This paper answers the question: which way should Monte Carlo Go go? Monte Carlo Go can clearly reduce statistical errors by collecting more samples. However, it is not yet known how much improvement can practically be achieved with additional samples. The contributions of this paper can be summarized as follows:

1. Examining the relationship between the sampling effort of Monte Carlo Go and playing strength in 9×9 Go. The existence of diminishing returns for additional samples is vividly observed through self-play experiments.
2. Demonstrating experimentally a relationship between samples and enhancements. Starting from the basic algorithm, two enhancements, *atari-50* and *progressive pruning* are incorporated. Improvements to the strength with these enhancements are discussed. In all versions, diminishing returns for additional samples is confirmed. Moreover, the gains of additional samples decline more quickly in the enhanced versions.
3. Analyzing move decisions in Monte Carlo Go with respect to the number of samples. The analysis suggests that Monte Carlo Go needs more samples to improve the quality of moves in the middle game.
4. Parallelizing Monte Carlo Go. This allows many more

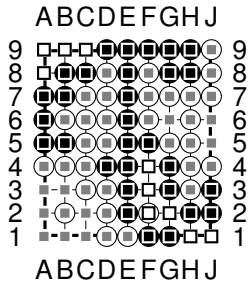


Figure 1: Example of a terminal position in Go

samples than in previous experiments (2,560,000 samples with 128 processors versus 10,000 samples for a practical compromise in (Bouzy & Helmstetter 2003)).

The structure of the paper is as follows: First, the rules of Go are briefly explained and related work is reviewed. Then, the implemented sequential and parallel Monte Carlo Go algorithm is presented, followed by experimental results. Finally, conclusions and future work are discussed.

The Game of Go

Go is popularly played especially in Asia. The rules are very simple. *Black* or *White* places a stone on an intersection point of a grid by turns. A pass is also legal. Although a board containing 19×19 points is usually used, 9×9 Go is also popular especially among beginners. In particular, a lot of researchers work on 9×9 Go because it still preserves many of the difficulties of this game.

A *block* is a directly connected set of stones of the same color. The empty points adjacent to a block are called *liberties*. A block is captured if the opponent plays on the last liberty of that block. A block is called *in atari* if it has only one liberty. In most variations of Go, except for moves capturing opponent blocks, a player is not allowed to make a move that results in any block of that player having no liberties. A single empty point surrounded by stones of the same block is called an *eye*. If blocks have two eyes, they are guaranteed to be *alive*. A *territory* is a surrounded area controlled by one player. Alive opponent stones cannot be contained in a territory. The game ends when the territories of each player are decided. To determine the winner, each player counts and compares the points in his or her territories.[1] For example, in Figure 1, points marked with grey squares are White's points and ones marked by white squares are Black's. In this figure, White wins because White has 41 points and Black has 40 points.

Related Work

This section reviews the literature on Monte Carlo Go and diminishing returns.

Monte Carlo Go

History Sampling-based approaches have been widely adapted to imperfect-information games (such as Bridge (Ginsberg 1999), Scrabble (Sheppard 2002), and Poker (Billings *et al.* 2002)). In perfect-information games, Abramson (1990) presented a method to use random sampling for evaluation. Applying the Monte Carlo method to Go was first introduced by Brügmann (1993), and was reinvestigated by Bouzy & Helmstetter (2003). Then, many enhancements have been presented including combination with tactical searches (Cazenave & Helmstetter 2005) or with Go knowledge such as patterns and move generators of existing Go programs (Bouzy 2005). Also, Bouzy combined Monte Carlo Go with d-ply search and a pruning method based on statistics of sampling at leaf nodes (2004). These efforts improve the strength of Monte Carlo Go programs reaching that of GNU Go, one of the best programs.

Basic Model and Enhancements Monte Carlo Go performs random sampling for position evaluation, instead of hand-coded evaluation functions. The basic model which we call the *one-ply-model* performs one-ply search and computes an "expected score" of each leaf. Then, it selects the move having the highest score. The expected score of a position is defined as the average of the final scores in the terminal positions of all random games starting from that position.[2] A fixed number of random games is played at each leaf. In a random game, each player almost randomly plays a legal move except for one filling in an eye point of that player[3] until the final score of the game is determined.

The *all moves as first heuristic* (AMAFH) is another model of utilizing random games. AMAFH selects the move having the largest difference between the *first-put-scores* for the player and that for the opponent. A first-put-score of a move for a player is defined as the average of final scores in the terminal positions of such random games that the player played the move before the opponent. AMAFH starts random games from the root, while the *one-ply-model* plays random games at leaves from depth 1. This property can reduce the number of samples (Bouzy & Helmstetter 2003).

Progressive Pruning is a pruning technique (Bouzy & Helmstetter 2003) in the *one-ply-model*. The idea is based on the fact that the average score of a larger number of random games can be statistically estimated by using the average score of a smaller number of random games. Let m_i be a mean value of move i, and σ_i be its standard deviation. Then, the mean value after a sufficient number of random games m'_i is expected to lie in the range $[m_i - \sigma_i r_d, m_i + \sigma_i r_d]$, where r_d is a confidence ratio. m_i is *statistically inferior* to m_j if and only if $m_i + \sigma_i r_d < m_j - \sigma_j r_d$ holds. After a sufficient number of random games, progressive pruning prunes a move as soon as it is statistically inferior to another move. Also, the sampling phase ends when all candidate moves become *statistically equal*, defined as follows: let σ_e be the standard deviation for equality, defined by experiments. Moves are *statistically equal* if and only if the standard deviation of each move is smaller than σ_e and no move is statistically inferior to other moves. As a result, pro-

[1] This paper assumes the Chinese rule with no *komi* (handicaps).

[2] Because of correct evaluation of terminal positions, a large number of samples can make the error rate of Monte Carlo Go evaluation close to 0, when it finds a winning/losing way.

[3] Filling one's eye is an extremely bad move in Go.

gressive pruning can reduce the number of samples required to select the best move.

Diminishing Returns in Search Algorithms

While diminishing returns for additional search is confirmed in many games (Billings & Björnsson 2003; Junghanns *et al.* 1997), demonstrating diminishing returns in chess took time. In self-play chess games, (Thompson 1982) showed that a program with $(d+1)$-ply search wins against the version performing d-ply search by a large margin. Surprisingly, no diminishing return was observed for additional search. Many follow-up experiments indicated that deeper search constantly improves the strength of chess programs (Berliner *et al.* 1990; Mysliwietz 1994). Although (Junghanns *et al.* 1997) did not also observe diminishing returns, they gave some evidence supporting the existence of diminishing returns. (Heinz 2001) finally demonstrated that diminishing returns appear with many games and deeper search performed by one of the strongest programs.

Diminishing Returns in Monte Carlo Go

On top of AMAFH, (Bouzy & Helmstetter 2003) compared Monte Carlo Go with 10,000 random games against versions with 1,000 and 100,000 random games. The difference in strength between 100 K and 10 K random games was smaller than that between 1 K and 10 K random games, while the difference in execution time between 100 K and 10 K random games was larger than that between 10 K and 1 K random games. He therefore concluded that 10 K random games is a good compromise in practice.

Statistically, the central limit theorem gives an insight to Monte Carlo Go. Assume that N random games are independent and let m be a mean of the random games and σ be the standard deviation. If N is large enough, the distribution of these random games approaches a normal distribution with mean m and standard deviation $\frac{\sigma}{\sqrt{N}}$. This indicates that more samples achieve less errors but the benefits of additional samples decrease. From a practical point of view, although Bouzy's experiments imply the existence of diminishing returns in Monte Carlo Go, there is no evidence that supports it. Moreover, no relationship between samples and enhancements has so far been investigated.

Implementation Designs for Monte Carlo Go

This section discusses sequential and parallel algorithms of our Monte Carlo Go implementation.

Sequential Algorithm

The techniques incorporated into our Monte Carlo Go program are summarized as follows:

Computation of Average Scores When implementing Monte Carlo Go, either the *one-ply-model* or AMAFH is so far available. However, because of drawbacks of AMAFH and intensive computation of the *one-ply-model* (Bouzy & Helmstetter 2003), a simpler approach is incorporated into our implementation: let s be the final score of a random game and move i be the first move in that random game. s is used to compute i's average score. This simplification can behave similarly to the *one-ply-model*, especially when a large number of random games are played for move decisions. Furthermore, it caused no degradation in strength against AMAFH.

Enhancements To investigate a relationship between enhancements and the number of random games, the following techniques are added:

- **Atari-50 Enhancement:** In a random game, each player plays a random move except for one that fills an eye point (Bouzy & Helmstetter 2003). This move selection scheme implies that adding Go-dependent knowledge is important in Monte Carlo Go. Since a move capturing stones is generally good in Go, the *atari-50 enhancement* increases the probability of such capture to 50%, instead of assigning a uniform probability to each move. Let N_1 be the number of capture moves, and N_2 be the number of other legal moves. Note that N_1 is almost always much smaller than N_2, because the number of capture moves is smaller than that of regular moves. If move i can capture stones, its probability is set to $\frac{50}{N_1}\%$. Otherwise, it is set to $\frac{50}{N_2}\%$.
- **Progressive Pruning:** A small modification is incorporated into our implementation, because our implementation does not perform one-ply search as in (Bouzy & Helmstetter 2003). Let move i be the first move of a random game. The final score of that random game is used to check if progressive pruning is applied to move i. This introduces differences in the number of samples among moves to apply progressive pruning. However, such errors become small with a large number of samples as in the previous subsection. Moreover, with the fixed number of samples at the root, our program can sample more frequently for unpruned moves, thus achieving more precise scores for such promising moves. In our implementation, at least 6,400 samples are required to turn on progressive pruning with $\sigma_e = 1.0$ and $r_d = 0.3$, tuned by many experiments.

Parallel Algorithm

Parallelizing Monte Carlo Go is a way to experiment with much larger sample sizes. Monte Carlo Go with no enhancement or the atari-50 enhancement can easily be parallelized, because no dependencies exist among random games. Let p be the number of processors and $N \times p$ be the number of samples. In our parallel implementation with no enhancement/atari-50, each processor locally collects N samples. Then, scores from all processors are combined to compute the expected score of each move.

Parallelizing Monte Carlo Go with progressive pruning must consider that some branches at the root are sometimes pruned. Our parallel algorithm consists of the master processor and a series of slave processors. As shown in Figure 2, the master gathers results from the slaves to check if progressive pruning can be applied. Each slave receives the pruning information from the master and plays random games.

Let p be the number of slave processors and $N \times p \times s$ be the total number of random games. The master splits the

```
/* Manage the average score μ and standard deviation σ. */
struct Stats { double μ, σ; };
Master() {
  /* μ and σ of each move is managed in stats. */
  struct Stats stats[NUM_LEGAL_MOVES];
  /* A set of flags checking if a move is pruned. */
  bool pruned[NUM_LEGAL_MOVES];
  /* No move is pruned in the beginning. */
  SetAllFlagsToFalse(pruned);
  for (i=0; i < s; i++) {
    BroadcastMsg(pruned, ALL_SLAVES);
    struct Stats temp[NUM_LEGAL_MOVES];
    /* Wait until all slaves end a stage of random games. */
    RecvAllSlavesResults(temp, ALL_SLAVES);
    Updateμ And σ(stats,temp);
    /* Select the best move with the largest μ. */
    best_move = GetMaxμMove(stats);
    m_l = stats[best_move].μ - r_d× stats[best_move].σ;
    foreach (move k) {
      m_r = stats[k].μ + r_d× stats[k].σ;
      /* Perform progressive pruning. */
      if( m_r < m_l) pruned[k] = true;
} } }
Slave() {
  /* Receive a set of flags to prune moves; */
  bool pruned[NUM_LEGAL_MOVES];
  struct Stats stats[NUM_LEGAL_MOVES];
  RecvMsg(pruned,MASTER);
  for (j=0; j < N; j++) {
    /* Randomly select the first move that is not pruned
       by progressive pruning. */
    move = SelectUnprunedMove(pruned);
    MakeMove(move);
    score = PerformRandomGames();
    UnmakeMove(move);
    stat[move] = Calcμ And σ(stat, move, score);
  }
  /* Send statistics to the master. */
  SendMsg(stat, MASTER);
}
```

Figure 2: Pseudo code of parallel progressive pruning

work into s stages. In each stage, the master asks each slave to play N random games. The master then tallies up results from all slaves to decide the moves to prune[4].

Although the algorithm waits for the last slave to finish N random games, it achieves very good load balancing. The execution time for each slave playing N random games is almost the same. The algorithm also has a trade-off between the number of stages and efficiency of progressive pruning. Large s incurs more communication overhead but can prune branches more frequently. On the other hand, although small s achieves less communication overhead, pruning occurs less frequently, causing slaves to play unnecessary random games. In this paper, $N \times p$ is always set to 6,400 so that the progressive pruning of the parallel algorithm is the same irrespective of the number of processors, and incurs small communication overhead.

Experimental Results

Programs and Environment

To measure the relative effectiveness of increasing samples, various experiments are conducted with three versions of Monte Carlo Go programs:

- BASIC: No enhancement is added,
- ATARI: The atari-50 enhancement is added to BASIC,
- ATARIPP: Progressive pruning is added to ATARI.

They were implemented with C++ and the MPI library (Gropp *et al.* 1996) for parallelism. The linear congruential method was incorporated into the implementation as a random number generator for efficiency, because no significant difference was seen in the strength of programs with Mersenne Twister in our preliminary experiments.

Experiments were measured in an environment that contains 192 PCs with an Intel Xeon Dual CPU at 2.40 GHz with 2 GB memory, connected by 1 Gb/s network. At most 64 PCs (i.e., 128 processors) were used for the experiments.

GNU Go[5] was used with the `aftermath` option to judge the winner of the two programs in each game.

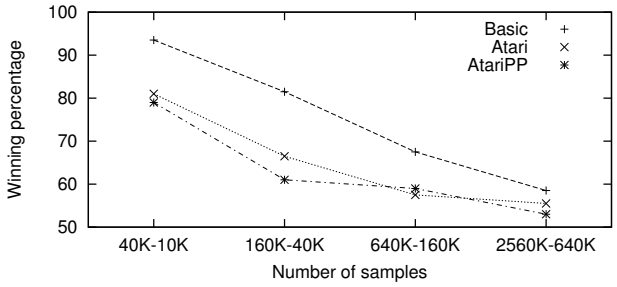


Figure 3: Self-play results with various sample sizes

Self-Play Results with Various Sample Sizes

First, self-play games with various numbers of samples were performed to show a relationship of diminishing returns between strength and sample sizes. Each program with N random games played a 200-game match against one with $N \times 4$ random games. Figure 3 shows the winning percentage, where N is plotted on the horizontal axis against the winning percentages for a program with $N \times 4$ samples on

[4]The master can obviously be a slave while the slaves play random games. However, our current implementation does not include this, because of the simplicity of the implementation and the large number of processors in our environment.

[5]http://www.gnu.org/software/gnugo/gnugo.html

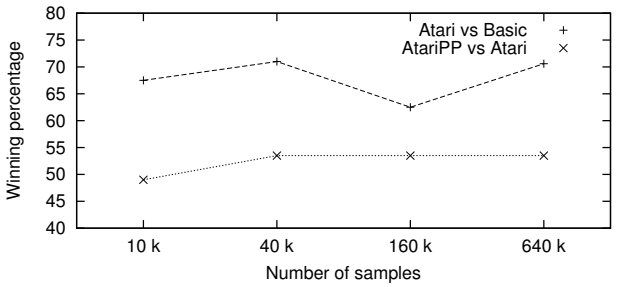


Figure 4: Self-play with enhancements

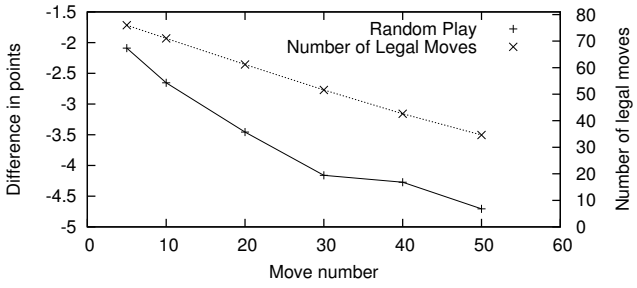


Figure 5: Score of the random player and the number of legal moves with progress of a game

the vertical axis. For example, 40K-10K on the horizontal axis indicates the match between a program with 40,000 and 10,000 random games. The result clearly shows that the benefits of additional samples decline as the sample size increases. Further, although the difference of diminishing returns between ATARI and ATARIPP is small, adding enhancements tends to make diminishing returns appear more quickly, reaching around the 55% winning percentage.

Performance when Adding Enhancements

Next, self-play games with the same sample sizes but with different enhancements were performed to show the relationship between the number of samples and the effectiveness of enhancements. Figure 4 shows results on the strength with additional enhancements and various sample sizes. The number of samples is plotted on the horizontal axis against the winning percentage of the version having more enhancements on the vertical axis.

Even if the sample size increases, ATARI consistently wins against BASIC, without decreasing the winning percentage. ATARIPP performs better than ATARI with large sample sizes. With the smaller number of samples, ATARIPP does not tend to exploit promising branches that cannot be found by ATARI, while ATARIPP occasionally prunes promising branches with some risk.

Decision Quality of Each Move

The moves selected by programs with various samples sizes were analyzed to estimate the relationship between the decision quality and the number of samples. 58 game records played by Japanese professionals were prepared[6]. At various points (moves 10 through 50) of each game, the expected scores of all legal moves were computed by using 64 million random games to create an *oracle*.

Figure 5 shows the number of legal moves and the quality of the random player against the BASIC oracle. The move number is plotted on the horizontal axis against the number of legal moves and the quality of the random player on the vertical axis. As the game progresses, the number of legal moves gradually decreases. Starting from 82 moves including a pass, it ends with around 30 moves. The quality of the random player is defined as the average of differences between the scores of oracle's best moves and the scores of the moves randomly chosen. Note that these scores are computed by oracle. The quality of the move chosen by the random player almost linearly decreases, where it is about -2 points in the opening and is about -5 points near the endgame. Results suggest that Monte Carlo Go (i.e., oracle in this case) tends to discover a larger difference between the best move and other moves as the game progresses.

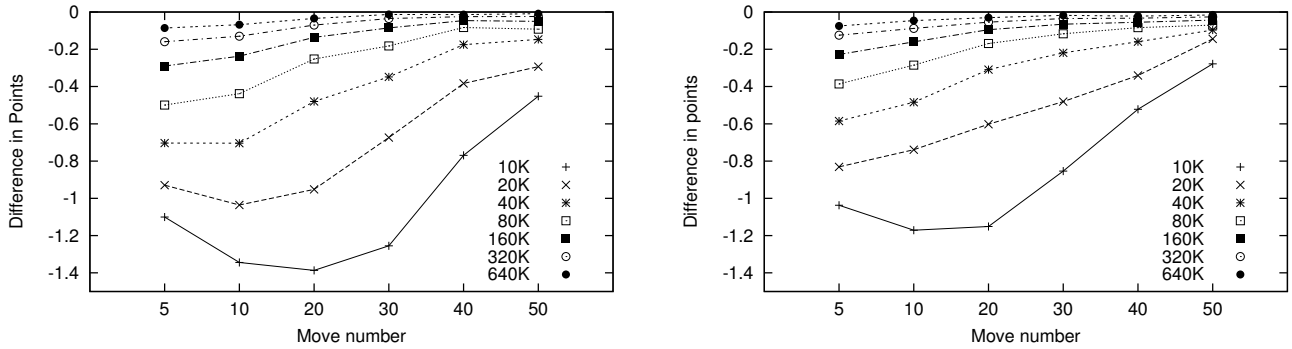
Figure 6 shows the quality of Monte Carlo programs with various samples sizes. The left figure (a) is for BASIC and the right one (b) is for ATARI. As in the previous figure, the move number is plotted on the horizontal axis against the quality of the selected move on the vertical axis. The oracle is defined as the version with the same enhancements and 64 million samples. In this figure, the quality approaches that of the oracle as the number of samples increases. Also, the quality difference decreases with enhancements. Again, these results confirm that the benefits of additional samples decline as the sample size increases and that additional enhancements make diminishing returns appear more quickly.

Results in Figure 6 also suggest a way of time control for Monte Carlo Go. The number of samples is usually fixed as proportional to the number of legal moves in a position (Bouzy & Helmstetter 2003), collecting less samples as the game progresses. However, the number of samples can be varied based on the stage of a game. For example, our results suggest that more samples must be collected in the middle games (moves 10 through 30).

Concluding Remarks

Bouzy & Helmstetter (2003) mentioned, "We believe that, with the help of increasing power of computation, this approach is promising for computer Go in the future." However, our experimental results demonstrate diminishing returns with additional samples, a problem that similarly affects the benefits of game-tree search for additional depth. Our results indicate strong evidence that advances in hardware technologies cannot constantly improve the performance of Monte Carlo Go, even if additional computational power enables them to sample more random games. A question on the best way to improve the strength of Monte Carlo Go is raised in this paper. One way is to concentrate more effort on enhancements such as atari-50 and progressive pruning.

[6]They are available at `http://gobase.org/9x9/book4`.



(a) BASIC (without enhancement) (b) ATARI (with Atari-50 enhancement)

Figure 6: Analysis of each move

Bouzy's recent research (2004) that combines Monte Carlo Go with minimax search and progressive pruning can be regarded as one way to avoid diminishing returns for additional samples. This approach raises a question on relations between sample sizes and search. Additionally, the expected scores of Monte Carlo Go basically have different characteristics from minimax values of game-tree search. So far, it is believed that search is excellent for tactics, while Monte Carlo is promising for evaluating strategic positions. Examining the (dis)advantages of both approaches will be a possible extension of this paper. Finally, because Monte Carlo sampling is a domain-dependent idea, extending this research to other domains will be an interesting topic.

Acknowledgement

We would like to thank Ian Frank, Markus Enzenberger, and Ling Zhao for beneficial feedback to the paper.

References

Abramson, B. 1990. Expected-outcome: A general model of static evaluation. *IEEE Trans. Pattern Analysis and Mach. Intell.* 12(2):182–193.

Berliner, H. J.; Goetsch, G.; Campbell, M. S.; and Ebeling, C. 1990. Measuring the performance potential of chess programs. *Artificial Intelligence* 43(1):7–20.

Billings, D., and Björnsson, Y. 2003. Search and knowledge in Lines of Action. In *Advances in Computer Games. Many Games, Many Challenges*, 231–248. Kluwer.

Billings, D.; Davidson, A.; Schaeffer, J.; and Szafron, D. 2002. The challenge of poker. *Artificial Intelligence* 134(1-2):201–240.

Bouzy, B., and Helmstetter, B. 2003. Monte Carlo Go developments. In *Advances in Computer Games. Many Games, Many Challenges*, 159–174. Kluwer.

Bouzy, B. 2003. The move decision strategy of Indigo. *ICGA Journal* 26(1):14–27.

Bouzy, B. 2004. Associating shallow and selective global tree search with Monte Carlo for 9x9 Go. In J. Herik, Y. Björnsson, N. N., ed., *4th Computer and Games Conference (CG04)*, LNCS 3846/2006, 67–80.

Bouzy, B. 2005. Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences* 175(4):247–257.

Brügmann, B. 1993. Monte Carlo Go. Technical report, Physics Department, Syracuse University.

Cazenave, T., and Helmstetter, B. 2005. Combining tactical search and Monte-Carlo in the game of Go. In *Symposium on Computational Intelligence and Games*, 171–175. IEEE.

Coulom, R. 2005. Crazy Stone. `http://remi.coulom.free.fr/CrazyStone/`.

Ginsberg, M. L. 1999. GIB: steps toward an expert-level bridge-playing program. In *Sixteenth International Joint Conference on Artificial Intelligence*, 584–589.

Gropp, W.; Lusk, E.; Doss, N.; and Skjellum, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22(6):789–828.

Heinz, E. A. 2001. New self-play results in computer chess. In *Second International Conference on Computers and Games (CG'2000)*, LNCS 2063, 262–276. Springer.

Junghanns, A.; Schaeffer, J.; Brockington, M.; Björnsson, Y.; and Marsland, T. 1997. Diminishing returns for additional search in chess. In *Advances in Computer Chess 8*, 53–67.

Müller, M. 2002. Computer Go. *Artificial Intelligence* 134:145–179.

Mysliwietz, P. 1994. *Konstruktion und Optimierung von Bewertungsfunktionen beim Schach*. Ph.D. Dissertation, University of Paderborn.

Nau, D. 1980. Pathology on game trees: A summary of results. In *First National AI Conference*, 102–104.

Sheppard, B. 2002. World-championship-caliber Scrabble. *Artificial Intelligence* 134(1-2):241–275.

Thompson, K. 1982. Computer chess strength. In Clarke, M., ed., *Advances in Computer Chess 3*, 55–56. Pergamon Press.