Monte-Carlo Strategies for Computer Go

Guillaume Chaslot [a] Jahn-Takeshi Saito [a] Jos W.H.M. Uiterwijk [a]

Bruno Bouzy [b] H. Jaap van den Herik [a]

[a] *MICC/IKAT, Universiteit Maastricht, The Netherlands*
[b] *CRIP5, Université René Déscartes, Paris, France*

Abstract

The game of Go is one of the games that still withstand classical Artificial Intelligence approaches. Hence, it is a good testbed for new AI methods. Amongst them, Monte-Carlo led to promising results. This method consists of building an evaluation function by averaging the outcome of several randomized games. The paper introduces a new strategy, which we call Objective Monte-Carlo, to improve this evaluation. Objective Monte-Carlo is composed of two parts. The first one is a move-selection strategy that adjusts the amount of exploration and exploitation automatically. We show experimentally that it outperforms the two classical strategies previously proposed for Monte-Carlo Go: Simulated Annealing and Progressive Pruning. The second part of our algorithm is a new backpropagation strategy. We show that it gives better results than Minimax in this context. Finally we discuss the extension of this method to other problems.

1 Introduction

Go is a 4000 years old game with simple rules. Nowadays, it is played by more than 40 million people, mostly in Asia, and there are hundreds of Go professionals. In Japan, China, and Korea, several TV channels and newspapers are dedicated to this game. Due to its simple rules and its fascinating structure, it constitutes an ongoing challenge for AI researchers. Up to 40 programs compete on various servers on the Internet, and during the Computer Games Olympiad, Go programs form the second-largest group after Chess. Despite the simplicity of the game, and the research effort, computer programs remain weaker than in other board games [14]. The reasons for this are: (i) the complexity of the evaluation function, and (ii) the size of the search space.

1.1 Search and Evaluation

The classical search paradigm for computational games is based on two tools: an evaluation function, and the Minimax back-propagation rule. The aim of these tools is to perform as much pruning as possible, e.g., as in $\alpha - \beta$ search [15] and Proof-Number Search [11]. Using these techniques, strong programs have been made for games like Checkers, Draughts, Othello, and Chess. In Go, all existing evaluation functions are highly biased. Hence, the classical concept of evaluating at the leafs of a tree does not work as well as in other games. Therefore, whereas research on the game of Chess has concentrated on the way of expanding the game tree, in the game of Go the quality of the evaluation function has a more important role. Monte-Carlo evaluation functions are shown to lead to rather precise evaluations and therefore are used in many programs.[1] They consist of playing random games from a position and inferring the value of this position from these games. In this paper we propose and test a move-selection and a backpropagation strategy which both improve the quality of Monte-Carlo evaluations.

[1] INDIGO, the Go program by Bruno Bouzy, was ranked 3^{rd} out of 9 in the Computer Olympiad 2005 in Taiwan. VIKING, OLEG, VEGOS, GOLOIS, CRAZYSTONE, GO81, and MANGO are other examples of Monte-Carlo Go programs.

1.2 Structure of the article

In Section 2, we present the structure of Monte-Carlo Go architectures, and algorithms previously used to improve them. In Section 3, we introduce Objective Monte-Carlo. In Section 4, we report on the experiments and their results. Section 5 contains the discussion. Finally, Section 6 concludes and proposes ideas for future research.

2 Related work

In this section, we first present the canonical structure of Monte-Carlo Go algorithms. Then we underline research ideas that have been developed to improve them. Finally we describe the main related algorithms that have previously been applied.

2.1 Monte-Carlo Architectures

In computer games, one possible Evaluation Function (EF) is the Monte-Carlo (MC) method. Given a board position B, its aim is to compute a value $V(B)$ for this position. Starting from the position B, MC plays a certain number of *simulated games*. A simulated game is a succession of moves (called *simulated moves*), played until the end of the game is reached.[2] The MC evaluation $V(B)$ is then deduced from the results of all the simulated games. In the simplest version, simulated moves are random moves, and $V(B)$ is the average of the outcomes of the simulated games.[3] This approach leads to a rather simple program. This program only uses the rules of the game but is nevertheless stronger than human beginners.

2.2 Research on Monte-Carlo Evaluation

The aim of research on MC evaluation is to improve the level of this method by different techniques. We distinguish three kinds of enhancements:

1. Integration of *domain-dependant knowledge* (DDK) in simulated games. Examples of DDK in the game of Go are patterns, tactical goals, or local heuristics [13, 7, 9]. To improve the quality of the simulated games, random simulated games can be replaced by *pseudo-random games* in which moves are chosen partially at random and partially using DDK. A typical implementation gives *DDK-weights* to each move [1]. In the simulated games, the probability of playing a move is the weight of this move divided by the sum of the weights of all possible moves. In earlier research, DDK-weights have been set manually. Lately research has concentrated on learning these weights automatically [5]. Similar techniques are used in Poker [8], Scrabble [16], Bridge [17], etc. This method of enhancement will not be discussed any further in this article, which will focus on the two other methods.
2. One can use the results of previous simulations to choose the move played in subsequent games. The basic idea consists of selecting the move that has shown to lead to the best results more often. This idea was first developed for the game of Go by Brügmann [6]. He used Simulated Annealing (SA) [10] to build a move ordering. Due to lack of computational power at the time, he made an approximation called "All-move-as-first Heuristic", which attributed a value to each move independently of the order in which it was played. This idea was taken over by Bruno Bouzy, who applied Progressive Pruning (PP). He showed in [2] that the All-move-as-first Heuristic was not useful anymore with the increase of computational power. In [3], he used PP in a depth-N context with N stages of evaluation and $N-1$ stages of selection. On the contrary, the algorithm that we propose does not make any distinction between a stage of selection and a stage of evaluation, but a smooth transition from evaluation to selection. Classical ways to use this kind of information are discussed further in this section. Our algorithm is described in detail in subsections 2.3.

[2] It is possible to develop different approaches. For instance, SlugGo does not play until the end, but only 16 simulated moves. Then it calls the EF of the open-source program GnuGo. This EF is quite slow. Furthermore, the simulated moves are played using GnuGo, which slows down the whole process. In fact, SlugGo only acts as a meta program. For these reasons, SlugGo needs a cluster of PCs to run.

[3] Some programmers use the average of the final scores, others use the winning percentage. It seems that many factors impact this choice. In our program, the average of the final scores leads to a better program. Hence, all the experiments made here use this criterium.

3. We show in this article that it is also possible to improve the backpropagation function. This issue will be discussed in Section 3.

2.3 Move-selection Strategies

The idea behind these strategies is to increase the ability of the Monte-Carlo evaluation to look ahead by exploring the more promising moves first. Before describing the strategies, a statistical background is required.

Each game i that is played has a result R_i. Let R be the random variable which takes the values R_i. The R_i values are bounded, so R has an average value μ and a standard deviation σ. In the case of 9×9 Go, $R_i \in [-81, 81]$ and σ is usually lower than 40.

The Central Limit theorem states that the standard deviation of the random variable $m_n = \frac{\sum_{i=1}^{n} R_i}{n}$ approaches $\frac{\sigma}{\sqrt{n}}$ when n approaches ∞. Furthermore, the probability distribution of m_n approaches the normal distribution $N(\mu, \frac{\sigma}{\sqrt{(n)}})$. Thus, a confidence interval for m_n can be deduced. Let m_∞ be the value of the average that we would expect after an infinite number of stochastic games. There is 66% of confidence that m_∞ is within the interval $[m_n - \frac{\sigma}{\sqrt{n}}, m_n + \frac{\sigma}{\sqrt{n}}]$, 95% of confidence that m_∞ is within the interval $[m_n - 2 \cdot \frac{\sigma}{\sqrt{n}}, m_n + 2 \cdot \frac{\sigma}{\sqrt{n}}]$, etc. Progressive Pruning and Objective Monte-Carlo are based on the results provided by this theorem.

2.3.1 Progressive Pruning (PP)

Progressive Pruning (PP) is a strategy used in the Go program of Bruno Bouzy, INDIGO [4]. It first associates a confidence interval to each move. The lower bound of this interval is $V - \frac{\sigma \cdot A}{\sqrt{n}}$, and its upper bound is $V + \frac{\sigma \cdot A}{\sqrt{n}}$, where V is the current evaluation of this move, n is the number of games simulated from this move, σ is the standard deviation on the value of one simulated game, and A is a parameter. In the beginning, every possible move is played the same number of times. For each move m, when the upper bound of the confidence interval of move m is below the lower bound of the best sibling move, m is pruned. The size of the confidence interval is in proportion to $\frac{\sigma}{\sqrt{n}}$, so when n approaches ∞, every move is pruned, exept the best move. When A is small, the algorithm is fast, because it prunes quickly several moves. However, it might also prune the best move. When A is large, the probability that it prunes the best move decreases strongly, but the algorithm becomes slower.

2.3.2 Simulated Annealing (SA)

This strategy, derived from physics [10], has been used for the first time in the field of Computer Go by Brügmann [6]. Its main conception is the following. In each step, a random action is carried out. This action changes a certain energy function E. Let ΔE be this change. If $\Delta E < 0$ the action is kept, otherwise it is kept only with the probability $e^{\frac{-\Delta E}{T}}$, where T is the current temperature. The strategy starts with a high temperature, which is then lowered. In the context of computer Go, each action is playing one move. If the energy function is the current evaluation of the move, then SA works as follows:

- A random move is made.
- If it is considered as the current best move, it is kept, and the simulated games continues.
- Otherwise there is a probability $e^{\frac{-\Delta E}{T}}$ to keep it, and a probability of $1 - e^{\frac{-\Delta E}{T}}$ to undo it. That means that the probability to keep the move is high if its current evaluation is near the evaluation of the best move.

The best annealing schedule has to be found experimentally.

3 Objective Monte-Carlo

Objective Monte-Carlo (OMC) is composed of two parts: a move-selection strategy, and a backpropagation strategy. In this section, we describe in detail these two new strategies. They will be tested independently in the next section.

3.1 Move-selection Strategy of OMC

The idea behind the move-selection stage of OMC is to exploit as much as possible the information provided by the Central Limit Theorem. We first give the pseudo code (see Algorithm 1), and then explain its main conception.

Data: Previous simulated move
Result: Next simulated move
$O_{bj} \leftarrow value\_of\_the\_current\_best\_child$
foreach *child move* $m \in M$ **do**
  $V_m \leftarrow current\_value\_of\_move\_m$
  $\sigma_m \leftarrow standard\_deviation\_of\_V_m$
  $U_m(O_{bj}) \leftarrow Erfc(\frac{O_{bj}-V_m}{\sqrt{2}\cdot\sigma_m})$
end
foreach *child move* $m \in M$ **do**
  $P_m \leftarrow \frac{U_m(O_{bj})}{\sum_{i\in M} U_i(O_{bj})}$.
end
Choose the next simulated move randomly according to the probability P_m for $m \in M$.

Algorithm 1: Move-selection Strategy of Objective Monte-Carlo

In this algorithm $Erfc$ stands for the complementary error function, i.e.,

$$Erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-u^2} du$$

For each possible score S, the Central Limit Theorem provides an approximation of the probability that the MC evaluation is S. Let $D_m(S)$ be the density function that gives the probability of a move m to have the average value S. $D_m(S)$ has the normal distribution $N(V_m, \frac{\sigma}{\sqrt{n}})$, where V_m is the current evaluation of the move m, and σ is the standard deviation of V_m.

Let O_{bj} be an objective score that the player aims to achieve. Using the distribution $D_m(S) = N(V_m, \frac{\sigma}{\sqrt{n}})$, we can compute the probability of the move m to be superior to O_{bj}. Let us call this probability $U_m(O_{bj})$. We have:

$$U_m(O_{bj}) = \frac{\sum_{x=O_{bj}}^{+\infty} D_m(x)}{\sum_{x=-\infty}^{+\infty} D_m(x)}$$

In Objective Monte-Carlo, $U_m(O_{bj})$ is considered as the *urgency* of a move, its importance. Thus the probability to chose the move is calculated according to these values, as shown in Algorithm 1. As can be seen, we approximate $U_m(O_{bj})$ by the $Erfc$ function. This function would be too long to compute in real time, hence the $Erfc$ values are stored in a table.

3.2 Backpropagation strategy of OMC

In this section we discuss how to build a backpropagation function better than Minimax for the MC context. This backpropagation strategy gives the value of move m according to the values and standard deviations of its children (see Algorithm 2). In a MC environment, when few simulations have been made, all the values calculated for the children are nearly random. Thus, the Minimax values measured for all children is the max of random numbers. Thus, it gives an overestimated value of the position. To avoid this error, the value returned by the backpropagation strategy should be close to the average value of the children in the beginning of the experiments. When the number of simulations made tend to infinity, on the contrary the value measured are perfectly accurate. So the context is exactly a Minimax context. Thus, when more simulations are made, the value returned should tend to the Minimax value.

In the beginning, all the urgencies U_c are nearly equal. So V_m can be approximated by:

$$V_m = \frac{\sum_{p\in M} V_p \cdot N_p}{\sum_{p\in M} N_p}$$

Therefore, in the begining the value of the move m is the average value of its children.

Data: Values of the children of m
Result: Value of m
foreach *child move* c *of* $m \in M$ **do**
  Compute the value of $U_c(Obj)$ according to Algorithm 1.
end
$N_c \leftarrow Number\_of\_games\_played\_with\_move\_c$
$V_m = \frac{\sum_{c \in M} U_c \cdot V_c \cdot N_c}{\sum_{c \in M} U_c \cdot N_c}$
return V_m

Algorithm 2: Backpropagation Strategy of Objective Monte-Carlo

Moreover, when the number of moves tends to infinity, $U_p(O_{bj})$ tends to $\frac{1}{2}$ if p is the best move, and to 0 else. [4] Thus this backpropagation tends to a Minimax backpropagation when the number of games played tends to infinity.

As a conclusion, this backpropagation strategy makes a soft transition from the average value to the Minimax value. This formula can only be applied in the case of Objective Monte-Carlo, where it is possible to compute the probabilities U_c.

4 Experiments

The aim of the experiments is to evaluate the strategies discussed in the previous section. In the first experiment we evaluate the impact of the move-selection strategy, and in the second experiment the backpropagation strategy. We have randomly selected one hundred positions from games of human players. The level of the selected players is about the same as the level of computer programs. This level was chosen to use positions as close as possible to the positions which the program will actually encounter.

4.1 Experimental setup

For each position, we first calculate an accurate evaluation of this position at depth one. To that prospect, we develop a tree at depth one, and then from each leaf launch a classical Monte-Carlo evaluation. This classical program will be called MC-Minimax. The number of stochastic games played is $200,000$ per leaf node[5]. The standard deviation of the score for a game of Go is below 40, so, according to the Central Limit theorem, the standard deviation on the MC-Minimax's value is less than 0.1 point. As it is an accurate evaluation of the position, in this paper we call this value the "real value" of the position.

Then, all the strategies are launched on all positions. For each pair of a position and a strategy, we consider the evaluation that the strategy gives minus the "real value" of the position. Finally, for each strategy we average over all positions. Hence, the value 0 represents a perfect evaluation of all positions, and positive values represent the average errors.

With time settings used in tournaments, we find that the maximal number of simulated games that can be performed for one evaluation with a 3GHz processor is less than $20,000$. Therefore this value is used as the maximum number of simulated games.

4.2 Results: Move-Selection Strategies

In this subsection, move-selection Strategies are evaluated at depth one. For all these strategies, the value returned is the Minimax value of the child nodes. The results are reported in Figure 1.

Several parameters had to be tuned. For SA it was important to select an adequate annealing schedule. If the annealing is too fast, the strategy does not converge to the real value of the position. If it is too slow then it converges to an accurate limit but very slowly. We tested 16 different annealing schedules and reported the best one (balancing speed and accuracy of convergence). For PP, a size for the interval of confidence has to be chosen. We used the one which is used in Bruno Bouzy's program, INDIGO, because it has been carefully tuned. If this

[4] This can be deduced from Algorithm 1.

[5] The average branching factor of the 100 games of the testbed is arround 50. The number of simulated moves played per game is arround 120. Hence the total number of simulated moves played to build this testbed is $200,000 \times 50 \times 100 \times 120 = 120 billions$. This required a couple of days on an Opteron 2.8GHz.

confidence interval is too small, it can happen that good moves are pruned, and the strategy does not converge to 0. If the confidence interval is too big, the convergence will be slower.

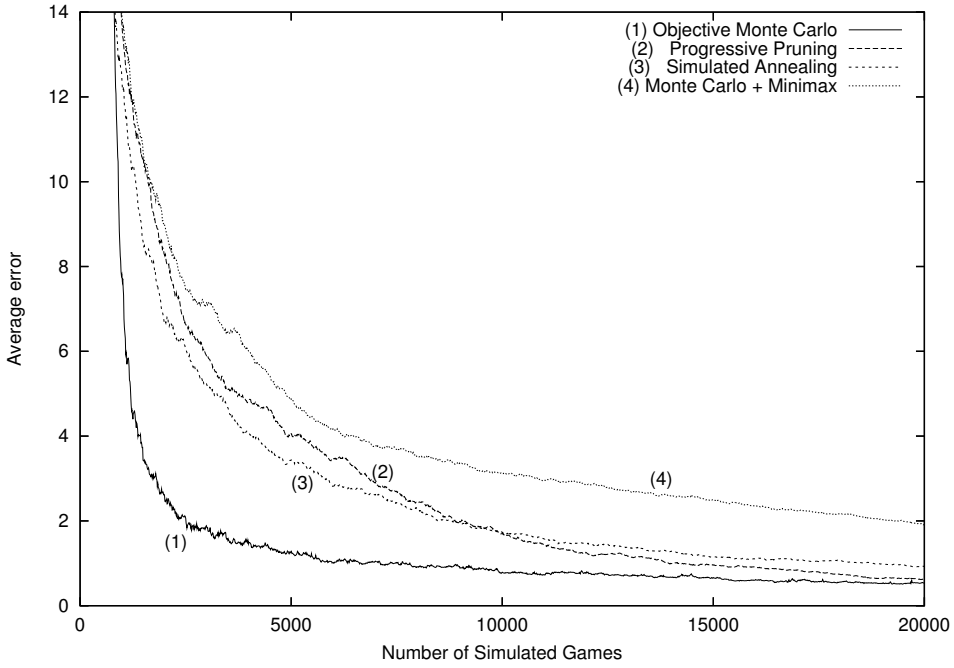

Figure 1: Convergence of Simulated Annealing, Progressive Pruning and Objective Monte-Carlo

The OMC move-selection strategy converges faster than the other strategies. With the parameters used, PP converges slower but to an accurate limit. SA (resp. PP) could converge faster with a faster annealing schedule (respectively a smaller confidence interval). However then they would converge to a worse limit because in some situations the best move would be pruned. OMC, on the contrary, never prunes any move.

4.3 Results: Backpropagation Strategies

We compare OMC backpropagation with Minimax backpropagation. The results are shown in Figure 2. The first thing to note is that OMC backpropagation improves considerably on the results when few simulations have been made. Furthermore, OMC Backpropagation is more accurate than Minimax backpropagation whatever the number of simulated games performed.

5 Discussion

In this section, we first point out the caracteristics of Monte-Carlo methods compared to classical methods. Then we focus on the specificity of our algorithm. Finally we discuss how to apply our strategy to other problems.

Monte-Carlo methods have the drawback to be slow. Indeed, the number of simulated games to be performed is important. Thus, Monte-Carlo programs are only able of performing shallow tree searches [3]. However, they are known to be particulary robust [2], so their lack of ability to look ahead is balanced by their ability to avoid global blunders. In this context, increasing the speed of convergence and building a new suitable tree search constitute two milestones.

To that perspect, OMC brings several improvements. Mostly it converges faster than usual strategies, and does not require any parameter to be tuned. In addition it presents two other advantages. First, this strategy can be applied in every node of the game tree. As it needs to store information about the values of the child nodes, the number of nodes in which it can be applied is proportional to the available memory. In our computer, we could apply it on 500,000 nodes, which corresponds to an average depth of three. This strategy is more efficient when it is applied near the root node since the gain is related to the number of simulated games

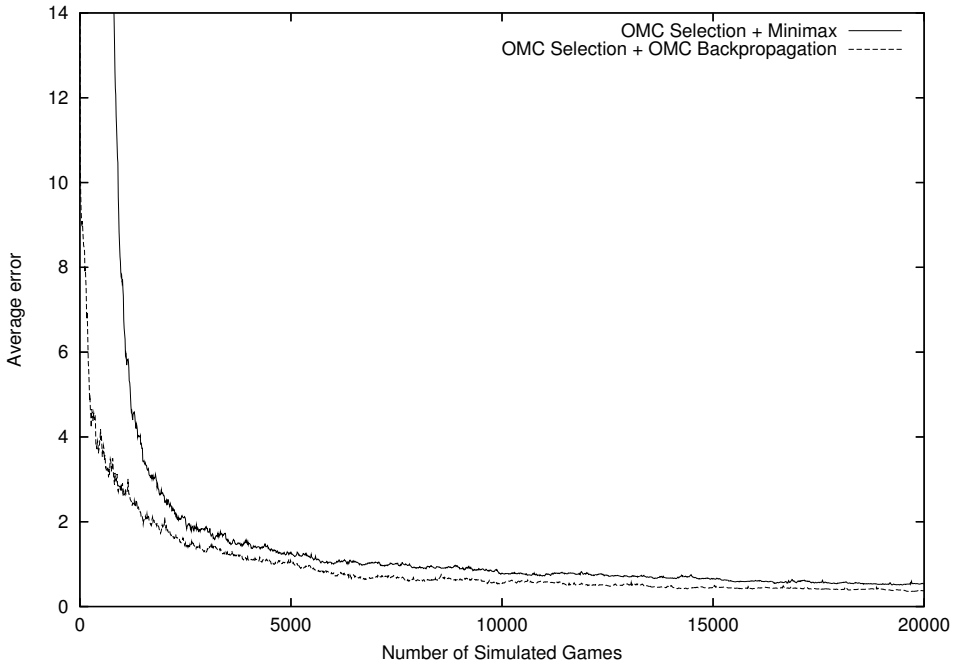

Figure 2: Convergence of OMC with Minimax backpropagation and with OMC backpropagation.

performed. We have shown the significant addition of our strategy at depth one. Our next research question will be to assess it at depth N for $N \geq 2$.

Another advantage of the OMC strategy is that it can adapt its style of play depending on who is winning. This can be done by adjusting O_{bj} slightly. If our program's EF evaluates that the program is winning, then it is not important to evaluate how much it is ahead. The crucial point is to avoid mistakes. The opponent must make a very good move to come back. Thus, the Monte-Carlo strategy should explore the "dangerous" moves more often. This can be achieved by increasing the objective O_{bj}. Thus, changing O_{bj} relative to the best successor node can make the program play in a more secure way, or in a more competitive way. This introduces a bias in the evaluation. However, this kind of bias can be useful, as the program would control it. For instance, if the program is winning it is better to have an overestimated evaluation of the moves of the opponent than an unbiased evaluation, because it induces a more secure style of play. Another research issue will be to find an algorithm that adjusts this bias automatically.

The framework we developed in this article can be applied to different kinds of adversarial problems, since it only uses simulations and statistics from these simulations. However, when it is possible to build an accurate evaluation function, as in most of board games, classical methods still perform better than Monte-Carlo ones. We believe that Real Time Strategy (RTS) games are in some cases a good candidate for Monte-Carlo methods, since it can be hard to build an accurate evaluation function. An application of Monte-Carlo methods in RTS can be found in [12].

6 Conclusion and future research

In this paper we introduced the Objective Monte-Carlo move-selection strategy, which outperforms the classical strategies, without any need to tune any parameter like an annealing schedule or a confidence interval. Indeed, the only parameter of this strategy, O_{bj}, is adjusted automatically. Then we proposed a backpropagation strategy better than Minimax for the OMC context. This algorithm has been implemented in our Go program, MANGO, and its level will be assessed in future computer Go tournaments.

The use of this selection strategy and the backpropagation strategy leads to a tree search that does not require any evaluation function. Hence, this can be applied to any game where it is difficult to create an evaluation function without parameter tuning. Assessing the ability of Objective Monte-Carlo to look ahead is our first topic for future research. Our next research

topic is to apply this strategy to other games than Go. Another research issue is to adjust automatically the value of the objective to take into account the fact that the program is winning or losing.

7 Acknowledgements

The authors would like to thank Steven de Jong for his support and adequate advice. This work is financed by the Dutch Organisation for Scientic Research (NWO) in the framework of the project Go for Go, grant number 612.066.409.

References

[1] Bruno Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Joint Conference on Information Sciences*, 2003.

[2] Bruno Bouzy. Monte Carlo Go Developments. In *Proceedings of the Advances in Computer Games conference (ACG-10)*, pages 159–174. Kluwer, H. Jaap van den Herik, Hiroyuki Iida, Ernst A. Heinz, 2003.

[3] Bruno Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9x9 Go. In K. Chen, editor, *Proceedings of 4th Computer and Games Conference, Ramat-Gan*, 2004.

[4] Bruno Bouzy. Move pruning techniques for Monte-Carlo Go. In *Proceedings of the 11th Advances in Computer Game conference*, page 15, 2005.

[5] Bruno Bouzy and Guillaume Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. In *IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA*, 2006.

[6] Bernd Brügmann. Monte Carlo Go. *White paper*, 1993.

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

[8] Jonathan Schaeffer Duane Szafron Darse Billings, Aaron Davidson. The challenge of poker. *Artificial Intelligence*, 134(1):201–240, 2002.

[9] Shun-Chin Hsu Jeng-Chi Yan and H.Jaap van den Herik. The sacrifice move. *ICGA Journal*, 28(4):223–234, 2005.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[11] Maarten van der Meulen L. Victor Allis and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.

[12] Michael Buro Michael Chung and Jonathan Schaeffer. Monte Carlo Planning in RTS games. In Simon Lucas and Graham Kendall, editors, *Proceeding of the IEEE Symposium on Computational Intelligence and Games*, 2005.

[13] Martin Müller. Pattern matching in Explorer: Extended abstract. In *Proceedings of the Game Playing System Workshop*, pages 1–3. ICOT, 1991.

[14] Martin Müller. Not like other games – why tree search in Go is different. In *Fifth Joint Conference on Information Sciences (JCIS)*, 2000.

[15] Judea Pearl. The solution for the branching factor of the alpha–beta pruning algorithm and its optimality. *Communications of the ACM*, 25:559–564, 1982.

[16] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1):241–275, 2002.

[17] Tom Throop Stephen Smith, Dana Nau. Computer Bridge - A Big Win for AI planning. *AI Magazine*, 19(2):93–105, 1998.