Combining Old-fashioned Computer Go with Monte Carlo Go

Florin Chelaru (Author)
Faculty of Computer Science
University "Al. I. Cuza" of Iasi
General Berthelot 16, 700483 Iasi, Romania
florin.chelaru@gmail.com

Liviu Ciortuz (Scientific Advisor)
Faculty of Computer Science - Computer Science Department
University "Al. I. Cuza" of Iasi
General Berthelot 16, 700483 Iasi, Romania
ciortuz@info.uaic.ro

Abstract

In this paper we discuss the idea of combining old-fashioned Computer Go with Monte Carlo Go. We introduce an analyze-after approach to random simulations. We also briefly present the other features of our present Monte Carlo implementation with Upper Confidence Trees. We then explain our approach to adding this implementation as a module in the GNU Go 3.6 engine, and finally show some preliminary results of the entire work, ideas for future work and conclusions.

1. Introduction

Go is one of the oldest games in the world, its origins going back long before its first historic references, which are dated in the 4th century B.C.

Its rules are few and simple, but its complexity exceeds by far those of other board games like Chess or Otello. While, in Chess, for instance, the number of legal moves available in an arbitrary state is a few dozens at most, the number of choices in Go may raise to a few hundred. This makes Computer Go one of the biggest challenges in artificial intelligence and computational theory. Moreover, the problem of finding an evaluation function is a difficult one, thus making it hard for programmers to use the classic artificial intelligence techniques like alpha-beta search in order to find good strategies. Consequently, the best Go-playing programs only reach average human level of performance.

The efforts for creating a good program for Go are commonly separated into two categories: old-fashioned Computer Go, referring to classic artificial intelligence techniques and strategies, and Monte Carlo Go, a recent approach based on probabilities and heuristics.

First studies and research in old-fashioned Go started about forty years ago focusing mostly on state representation, breaking down the game in goal-oriented sub-games, local searches and local results, functions for evaluation and determining influence and base knowledge for pattern matching. There are several programs using this approach, of which one of the best and the only one with available sources and documentation is GNU Go.

Since the beginning of 1990’s, and more intensely within the last 10 years, the attention has moved over some probabilistic approaches, the most important being Monte Carlo Go. Monte Carlo is a simple algorithm, based on approximating the expected outcome of the game. At each step, before playing a stone, the program launches a number of random simulations, starting with each available move, which are evaluated. The move with the best average score is picked as the best one available and played. Used along with various heuristics, it gave birth to impressive results.

It is estimated that, on 9×9 boards, for a one point precision evaluation, 1000 random simulations give 68% statistical confidence, while 4000 games 95%. Present CPU’s are able to compute about 10,000 random simulations per second, which means that the method works in reasonable time.

1The first Go program was written by Albert Zobrist in 1968 as part of his thesis on pattern recognition ([1]).
3The general Monte Carlo model was advanced by Bruce Abramson, who used it on games of low complexity, such as 6×6 Otello ([7]). In 1993, Bernd Brügmann created the first 9×9 MC Go program, Gobble ([8]).
and with enough statistical confidence on small boards\(^4\).

There are several arguments as to which of the two - the old-fashioned and the Monte Carlo approaches is more appropriate to handle the problem of Computer Go. In this article, we propose a way of combining both of them into one strong application.

Our focus falls upon three aspects. In Section 2, we discuss about an algorithm we propose for Monte Carlo random simulations, having speed in the center of attention. Then, in Section 3, we briefly present our preliminary Monte Carlo implementation, with future plans of improvement. There, we go through the most important of the heuristics known and used so far and already proven to yield good results. Finally, in Section 4, we talk about adding a Monte Carlo module to the engine of GNU Go 3.6 and using it to enhance its results.

2. The Analyze-after approach to random simulations

Although, on small boards, Monte Carlo behaved very well, on the 19\(\times\)19 board, the results are not yet as satisfactory. This is due to the fact that both the average number of legal moves and the length of the game grow along with the size of the board. Particularly, both the time needed to compute a random simulation, and the number of random simulations needed to evaluate a state, increase. This made us look for a fast random simulation algorithm, which we call analyze-after.

The idea of the algorithm is to place stones on the board, not paying attention to any rules, except that of not filling friendly eyes. Then, when all the board is full, analyze it and decide which moves were illegal, which stones were captured, and finally, decide which territory belongs to whom. In order to present this approach, we will formalize the concepts in the game of Go, so it is easier to see the solution.

Let \(N\) the size of the board. Then \(B\), the set of intersections and \(C\), the set of colors, can be written as:

\[
B = \{(x, y) \mid 1 \leq x, y \leq N\} \text{ and } C = \{\text{black, white}\}.
\]

Also, the set of neighbors of an intersection \(i = (x, y)\) is

\[
N_i = B \cap \{(x + 1, y), (x, y + 1), (x - 1, y), (x, y - 1)\}.
\]

We introduce a general concept of game:

\[
G_{k, k \geq 1} = \left\{g_k \mid g_k : \{1, \ldots, k\} \rightarrow B \times C \right\}
\]

\(G_k\) is the set of all games with \(k\) moves. \(g_k(t) = (i, c)\) represents the \(t\)th move made in game \(g_k\). So \(t\) is the moment when a stone of color \(c\) was placed on the board at intersection \(i\). In other words, the domain of any \(g \in \bigcup_{j \geq 0} G_j\) represents the set of indexes for all moves played during the game, while the values in the codomain represent moves themselves. \(g(1)\), thus, represents the first move, expressed by a stone of a certain color placed at a certain intersection; \(g(2)\) is the second move, and so on.

We aim to find an algorithm that, based on a game \(g\), and respecting the rules of Go, retrieves the state of the board at the end of \(g\).\(^5\) In other words, the following properties should be satisfied:

1. Any group of stones on the board has at least one liberty;
2. The order in which the stones were placed on the board dictates which worms stay and which are removed (they were captured either by an enemy stone, or by a suicide move).

For an illustration of these properties, consider the sequence of moves in figure 1.

![Figure 1.](image)

The numbers inside the stones represent the indexes of the moves. The first property specifies that a board state containing this exact configuration is not allowed: the stones played \(4^{th}\) and \(7^{th}\) have no liberties. In consequence, at least one of them should be removed. The second property tells us that the order in which we should remove stones from the board should be the same in which the stones were played. In other words, if we were to choose which to capture between the \(4^{th}\) and the \(7^{th}\) stone, since the \(7^{th}\) was played later, thus making a capture, we remove the \(4^{th}\).

For convenience, we consider games where any intersection is played at most once. The results can be easily generalized. Formally, these games satisfy:

\[
g(t) = (i, c) \Rightarrow (\forall t' \neq t)(g(t') = (i', c') \Rightarrow i' \neq i) \quad (1)
\]

Theorem 1. There is an algorithm \(b\) that, for all \(g \in \bigcup_{j \geq 0} G_j\) with property (1), determines the state of the board at the end of \(g\), \(b_g : B \rightarrow C \cup \{\text{empty}\}\) satisfying 1. and 2.

\(^4\)See also [7].

\(^5\)We should mention that, for any sequence of moves there is a valid board state describing it.
Proof. We have to prove that, having a game \( g \), we can associate to it a set of pairs \((intersection, stone)\) or \((intersection, empty)\), for every intersection on the board, which represents the configuration after the last move in \( g \) was played.

Let \( \text{time}_{g_k} : B \to \{1, \ldots, k\} \) be defined as

\[
\text{time}_{g_k}(i) = \begin{cases} 
\min \{ t \leq k \mid \exists c \in C : g_k(t) = (i, c) \} \\
\infty, \text{ if no such } t \text{ as above exists},
\end{cases}
\]

for all \( g_k \in \bigcup_{j \geq 0} G_j \). \( \text{time}_g(i) \) represents the first moment during \( g \) when the intersection \( i \) was occupied by a stone of color \( c \).

We can now define a first notion of board state with no rules (i.e. not respecting properties 1. and 2.), \( \tilde{b}_g : B \to C \cup \{\text{empty}\} \):

\[
\tilde{b}_g(i) = \begin{cases} 
\emptyset, \text{ if } \text{time}_g(i) = (i, c) \\
\in C, \text{ otherwise},
\end{cases}
\]

for all \( g \in \bigcup_{j \geq 0} G_j \). \( \tilde{b}_g \) is the way a board would look like at the end of \( g \) if no rules were respected, thus no stones removed. We need this notion of board state as a point to start from, when applying the rules.

Now, the notions of \textit{worm}, and \textit{worm neighbors} come naturally. The worm, \( w_{g,i} \subseteq B \) is defined inductively, as:

**Base**

\[
w_{g,i} = \begin{cases} 
\{i\}, \text{ if } \tilde{b}_g(i) \in C \\
\emptyset, \text{ otherwise},
\end{cases}
\]

In other words, any stone on the board belongs to a worm containing at list that stone. If an intersection is empty, then it doesn’t belong to any worm.

**Inductive step**

Let \( i' \in w_{g,i} \).

If \( i'' \in N_{i'} \)

\[
\tilde{b}_g(i'') = b_g(i'),
\]

then, \( i'' \in w_{g,i} \).

So all the friendly neighbors of a stone belong to the same worm as the stone itself.

**Observation.** Notice that \( \forall i', i'' \in w_{g,i} \) satisfy \( w_{g,i'} = w_{g,i''} = w_{g,i} \). In other words, a worm may be determined by several stones. This is natural and intuitive, but still, we mention it for clarity.

The worm neighbors, \( N_w \subseteq B \), are defined below:

\[
N_w = \{ i'' \in B \mid \exists i' \in w : i'' \in N_{i'} \} \setminus w.
\]

In order to obtain the board, we use a function called \( \text{lateness}_g : \mathcal{P}(B) \to \mathbb{N} \) giving the time the latest stone of the worm was played:

\[
\text{lateness}_g(w) = \max \{ \text{time}_g(i) \mid i \in w \}.
\]

We use this function so that we can apply the second rule to the state of the board. In other words, when choosing which to remove between two worms, we will remove the oldest, i.e. the one with the lowest \( \text{lateness}_g(w) \).

We can now finally define the state of a board, at the end of the game, \( b_g : B \to C \cup \{\text{empty}\} \):

**Base**

\[
(\forall i \in B) \tilde{b}_g(i) = \text{empty} \Rightarrow b_g(i) = \text{empty}
\]

**Inductive step**

Let

\[
w = \arg\min_{w_{g,i}} \{ \text{lateness}_g(w_{g,i}) \mid w_{g,i} : b_g(i) \text{ undefined} \}.
\]

Then, \( \forall i' \in w \),

\[
b_g(i') = \begin{cases} 
\tilde{b}_g(i'), \text{ if } \exists i'' \in N_w : b_g(i'') = \text{empty} \\
\text{empty}, \text{ otherwise}.
\end{cases}
\]

So, at every step, we choose the oldest worm for which we haven’t established whether is should be removed or not. If the worm has liberties, it stays on the board. Otherwise, its intersections become \textit{empty}.

In the following lines, we prove, inductively, that the algorithm is correct.

Let \( b_g^* : B \to C \cup \{\text{empty}\} \) be the correct state of the board at the end of the game. We show that \( b_g^*(i) = b_g(i), \forall i \in B \).

**Base**

\[
b_g(i) = \text{empty} \Rightarrow \text{no stone was placed at } i \text{ during } g
\]

\[
\Rightarrow b_g^*(i) = \text{empty} \Rightarrow b_g^*(i) = b_g(i)
\]

**Inductive step**

We assume that, up to this step, \( \forall b_g(i) \text{ defined}, b_g^*(i) = b_g(i) \).

Suppose, by reduction ad absurdum, that \( \tilde{b}_g(i') \neq \text{empty} \), \( b_g^*(i') = \text{empty} \), and \( b_g(i') = \tilde{b}_g(i') \). Then, \( \exists i'' \in N_w : b_g(i'') = \text{empty} \). According to the inductive hypothesis, this implies that \( b_g^*(i'') = \text{empty} \), which means that \( w \) has at least one liberty, at \( i'' \), so there was no need for it to be removed during the game.

Now suppose that \( \tilde{b}_g(i') \neq \text{empty} \), \( b_g^*(i') = \tilde{b}_g(i') \), and \( b_g(i') = \text{empty} \). Taking account of the inductive hypothesis, this means that \( \forall i''' \in N_w : b_g(i''') \text{ is undefined} \). \( \Rightarrow \text{argmax}_{i'} \{ \text{time}_g\mid j \in w_{g,i'} \} \). This means that the first to lose its liberties between \( w \) and \( w_{g,i'} \) is \( w \). In other words, \( i' \) and the entire worm it belongs to, was removed from the board.

\( \square \)
Theorem 2. The algorithm retrieves, for each \( g \in \bigcup_{j>0} G_j \), the state of the board in \( O(|\mathcal{B}| + |\mathcal{W}| \log |\mathcal{W}|) \) time, where \( \mathcal{W} \) is the set of all worms on the board.

Proof. The algorithm can be written as the following sequence of steps:

1. Determine worms.
   - Iterate through the entire board to make an array containing, for every worm, a list with all stones.
   - For every worm, \( w \), store lateness\( g(w) \), which represents the time when the last stone in \( w \) was placed on the board, and
   - store its neighbors.
2. Sort worms by their lateness.
3. Remove captured worms.
   - Iterate through worms from oldest to newest.
   - If the worm has no liberties, remove it from the board, and
   - mark all its neighbors as having at least one liberty.

The first step has complexity \( O(|\mathcal{B}|) \), since it involves an iteration through the entire board. Sorting the worms takes \( O(|\mathcal{W}| \cdot \log |\mathcal{W}|) \). Removing captured worms takes \( O(|\mathcal{B}|) \). Summing up, we reach the desired result.

This result tells us that, having a sequence of completely arbitrary moves, we can generate a valid Go board state, as if the rules had been followed from the beginning. In other words, if we just placed stones on the board until there is literally no place to move, there would be a way to extract a correct Go endgame board, having clearly delimited territories, and the resulted board would be easy to evaluate.

3. Heuristics for Monte Carlo Go

Being a relatively new approach, research is still open on the subject of Monte Carlo applied to Go. Still, several heuristics have already shown themselves to give positive results, of which some more than others. In this section we discuss about the most important of them, which either we have implemented or we consider that should be implemented in a future version of our work.

3.1. Upper Confidence for Trees (UCT)

UCT represents an algorithm based on the Multi-armed Bandit problem\(^6\). The problem basically refers to the Exploration versus Exploitation dilemma, which consists of searching for a balance between exploring the environment to find profitable actions and taking the empirically best action as often as possible.

Formally, a \( K \)-armed bandit is represented by random variables \( X_{i,t} \), where each \( i \) is the index of a gambling machine, which on successive plays yields rewards \( X_{i,1}, X_{i,2}, \ldots \). The rewards are independent and identically distributed according to an unknown law with unknown expectation \( \mu_i \). Independence also holds for rewards across machines; i.e., \( X_{i,t} \) and \( X_{j,t} \) are independent for each \( 1 \leq i < j \leq K \) and each \( s, t \geq 1 \). The purpose of the gambler is to find a strategy of maximizing his winnings.

It has been proven that, choosing at each step the machine which maximizes the following formula (called UCB1-TUNED, or simply UCB1) ensures the play of the overall best machine exponentially more often than the others:

\[
\mathcal{X}_j + \sqrt{\frac{\ln n}{T_j(n)}} \cdot \min \left\{ 0.25, V_j \left( \frac{T_j(n)}{T_j(n)} \right) \right\}, \quad (2)
\]

where:

- \( T_j(n) \) is the number of times machine \( i \) has been played after the first \( n \) plays,
- \( \mathcal{X}_{i,s} = \frac{1}{s} \sum_{t=1}^{s} X_{i,t} \); \( \mathcal{X}_i = \mathcal{X}_{i,T_i(n)} \) and
- \( V_j(s) = \left( \frac{1}{s} \sum_{t=1}^{s} X_{j,t}^2 \right) - \mathcal{X}_{j,s}^2 + \sqrt{\frac{2 \ln n}{s}} \) (an estimated upper bound of the variance of machine \( j \)).

UCT consists of treating every node in the Monte Carlo tree as a bandit problem, all move choices representing the machines. At first, for each available move, a random simulation is launched. Then, at each step, the algorithm chooses the one for which (2) has the greatest value\(^7\) and plays the next random game starting with it. The rewards may be 1, if the game ended in favor of the color playing the first move or 0 otherwise.

3.2. All-moves-as-first

This heuristic has an important word to say for the Monte Carlo Go playing engines, since it allows the process of evaluating a move to divide the response time by the size of the board. The idea is simple: after a random game with

\(^6\)See [2] for a detailed description of the problem and the solution we use in our implementation of UCT.

\(^7\)See [11] for an extended presentation of UCT.
a certain score, instead of just updating the mean of the first move of the random game, the heuristic updates all moves played first on their intersections with the same color as the first move. It also updates with the opposite score the means of the moves played first on their intersections with a different color from the first move [5].

Basically, this updates the means of almost all moves in the game. Of course, the heuristic isn’t entirely correct, since various moves may have different effects on the game depending on the time they were played. Still, the speedup is worth taking into consideration.

3.3. Rapid Action Value Estimation (RAVE)

RAVE\(^8\) is a heuristic used for generalizing the value of a move across all positions in the subtree below a certain Go state. It is closely related to the all-moves-as-first idea. Moreover, it provides a way of sharing experience between classes of related positions. The strategy uses the following formula:

\[
\hat{Q}(s, a) = \frac{1}{\hat{n}(s, a)} \sum_{i=1}^{N} \hat{I}_i(s, a) z_i,
\]

where:

- \(z_i\) is the outcome of the \(i^{th}\) simulation: \(z = 1\) if the game was won and \(z = 0\) otherwise;
- \(\hat{I}_i(s, a)\) yields \(1\) if position \(s\) was encountered at any step \(k\) of the \(i^{th}\) game, and move \(a\) was selected at any step \(t \geq k\), or \(0\) otherwise;
- \(\hat{n}(s, a) = \sum_{i=1}^{N} \hat{I}_i(s, a)\) counts the total number of simulations used to estimate the RAVE value; and
- \(\hat{Q}(s, a)\) represents the average outcome of all simulations where move \(a\) was selected in the position \(s\), or in any subsequent position.

The idea is to use (3), which is biased but with lower variance at the beginning of the game, and gradually shift to the classic Monte Carlo value, unbiased and with higher variance.

3.4. Grandparent knowledge

Grandparent knowledge is an approach related to the two previous ones. It basically uses the fact that moves at the same intersection, which are close in time, have close influence over the outcome of the game. Let \(s\) be the current state and \(g\) be its grandparent node. Then, at the time \(g\) was the current state, Monte Carlo explored the uncles of \(s\), i.e. the alternative moves to the parent of \(s\), the one eventually chosen. The idea is to initialize the sons of \(s\) with the values of its uncles before any simulations are launched. Although the results of this technique are not spectacular, we consider it worth mentioning.

4. Integration of Monte Carlo within GNU Go

4.1. GNU Go engine overview

GNU Go starts by trying to get a good understanding of the current board position. Using the information found in this first phase, and using additional move generators, a list of candidate moves is generated. Finally, each of the candidate moves is valued according to its territorial value (including captures or life-and-death effects), and possible strategic effects (such as strengthening a weak group).

Although GNU Go does a lot of reading to analyze possible captures, life and death of groups etc., it does not have a full-board lookahead and this is the main point where improvements can be made\(^9\).

4.2. Adding a Monte Carlo module to GNU Go

We try to solve this latter problem of GNU Go, by adding the Monte Carlo module. The functionality of this module is as follows.

A separate thread runs random simulations during opponent time, exploring the UCT tree. Whenever a move is to be generated, the thread pauses, waiting for GNU Go’s engine to generate a list of moves. Each of the moves has associated reasons summing up to a value estimating how good it is. After the list is generated, Monte Carlo resumes for a given amount of time, so that the confidence of its evaluation is good enough. Then again, it pauses. At this point, every available move has an associated winning probability. The next step is intuitive. For every move with positive score in the list generated by GNU Go we take its value and multiply it by its winning probability. This way, moves considered good both by GNU Go and Monte Carlo are automatically chosen. Moreover, moves estimated by GNU Go to have the same local influence, which in fact have different global importance in the game are overall ranked accordingly by Monte Carlo. Also, inherent errors which appear in Monte Carlo-only applications, due to lack of local precision, are eliminated thanks to GNU Go’s old-fashioned Computer Go approach.

\(^8\)A detailed description of RAVE and integration within Monte Carlo and UCT is provided in [10].

\(^9\)For a full description of the GNU Go engine, visit the GNU Go documentation page: http://www.gnu.org/software/gnugo/gnugo_getoc.html
5. Preliminary results

Our application was implemented in ANSI C, respecting the coding style and conventions of GNU Go 3.6, specified in [3], Section 4.6. The sources were compiled under Microsoft® Windows XP® Service Pack 2, using Microsoft® Visual Studio® 2005 Team Suite. For testing we used a system with an AMD® Athlon® XP 2700+ (2.17 GHz) CPU. All tests are made on the 19×19 board.

5.1. The random simulation

We have tested the speed of random simulations, comparing the naïve and the analyze-after algorithms. Table 1, along with Figures 3 and 4 show a comparison of the results given by the two approaches. They clearly show the speedup of the analyze-after algorithm over the naïve one.

### Table 1. Random Simulations: Algorithm Performance Comparison. The statistics are made over 100 tests for each algorithm. The values represent average numbers of simulations per second.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Min</th>
<th>Max</th>
<th>μ</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>90.99</td>
<td>98.19</td>
<td>95.41</td>
<td>1.38</td>
</tr>
<tr>
<td>Analyze-after</td>
<td>1028.4</td>
<td>1205.2</td>
<td>1058.62</td>
<td>38.47</td>
</tr>
</tbody>
</table>

5.2. Monte Carlo GNU Go versus GNU Go

In the current section we present the outcomes of the games played by our program, called Monte Carlo GNU Go (or McGnuGo, for short) and GNU Go 3.6 (or GnuGo). These represent preliminary results, since the number of games tested so far is still small. However, we created a statistic based on this data, and we are able to explain different ups and downs of our approach, which resulted from these results. Also, we suggest some possible enhancements to our work, which, we are confident, will improve the performance of our application.

Table 2 shows the results of the games played so far.

### Table 2. McGnuGo against GnuGo

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win percentage as White</td>
<td>68.18%</td>
</tr>
<tr>
<td>Win percentage as Black</td>
<td>45.45%</td>
</tr>
<tr>
<td>Overall win percentage</td>
<td>60.60%</td>
</tr>
<tr>
<td>Average outcome (μ)</td>
<td>1.106</td>
</tr>
<tr>
<td>Standard deviation (σ)</td>
<td>24.54</td>
</tr>
</tbody>
</table>

6. Conclusions

The advantages of our approach are straightforward: we have obtained a better engine than the one we started from. However, we think that the results can be improved by handling various problems which arose.

The first thing that we noticed was that when playing with black, our program lost more games than it won, and especially, the results differed a lot from the ones recorded when playing with white. Another thing we noticed, was the fact that most of the times, when losing a little advantage at the beginning of the game, our engine started making bad moves, thus losing even more, eventually being defeated by a big difference. One last thing we were able to pick up from the tests so far is the fact that, whenever in a tactical situation, where the opponent is clearly in advantage,
our program tends to favor moves extending territory somewhere else on the board, ignoring the threat. This eventually leads to losing the game.

We try to explain these problems in the following paragraphs, presenting solutions which we intend to implement in our future work.

One of the most important issues and reasons for wrong decisions is the fact that, in many cases, when speaking about random simulations, \( \text{average outcome} \neq \text{actual outcome} \). In other words, most of the results yielded by random games are far from what even an average Go player would play, most of them having no sense at all. This makes the mean of the random simulations get far from the actual situation on the board when a tactical situation shows up, and only get close to it when a clear territorial advantage is present. This explains the fact that McGnuGo ignores, in most cases, the strategic threat situations.

A potential fix we suggest is to use a temperature variable, similar to the one in the Simulated Annealing algorithm, which, taking account of previous plays and current evaluations, favors the moves advised by the GnuGo valuation over the ones advised by the Monte Carlo module. This variable starts with a higher chance of deciding over the move for Monte Carlo, and during the game, adapts to the actual situation.

Another issue, which we consider worth speaking about, is one concerning the fact that the outcomes of the random simulations, as far as Monte Carlo is concerned, come in two values: 0, for defeat and 1 for victory, no intermediate values. Whenever our program loses some advantage, most of the random simulations start yielding 0, telling the program one thing: the game is most likely lost. So instead of trying to maximize its score, our program prematurely abandons hope. The same observation can be made when it is clearly in advantage, which causes McGnuGo to not make the difference between a potential bad move, and a better move.

The fix for this problem, we think, comes also by associating a temperature variable to Monte Carlo, helping it decide whether to make strict evaluations (0 and 1) or differentiated evaluations, depending on the actual score given by the simulation. When the situation is balanced on the board, the variable would lean onto the 0/1 evaluation, while when in clear advantage/disadvantage, it would try to choose the moves yielding better scores rather than just wins/loses.

The inaccuracy of random games, associated with the abandoning hope, also explains, in part, the fact that black loses more than white: when playing randomly, the advantage given by the fact that black places the first stone is
much smaller than the standard value of the $komi^{10}$. This makes Monte Carlo believe that the game is lost, and acts as described earlier.

A solution which partially solves the problem is setting a score average for the last number of random simulations and, if the next score is over the average, decide it to be a victory, while if it is below average, defeat. This should make the program constantly choose the better moves most of the time.

Another solution to the inaccuracy of the random games is implementing a pattern-matching approach. See, for further reference, [11], [12], [6] and [9].

Still, most of the times, our Go player is an offensive one, trying to expand territory as much as possible, and moreover, to invade and capture enemy large zones. This gives the good results observed in Table 2. The balance between GnuGo’s move choices and Monte Carlo’s enhancements is acceptable and overall, we consider this to be a first step towards an excellent Go player.

References


$^{10}$We used $komi = 6.5$. 