MONTE-CARLO GO DEVELOPMENTS

B. Bouzy
Université Paris 5, UFR de mathematiques et d'informatique, C.R.I.P.5,
45, rue des Saints-Pères 75270 Paris Cedex 06 France
bouzy@math-info.univ-paris5.fr

B. Helmstetter
Université Paris 8, laboratoire d'Intelligence Artificielle
2, rue de la Liberté 93526 Saint-Denis Cedex France
bh@ai.univ-paris8.fr

Abstract We describe two Go programs, OLGA and OLEG, developed by a Monte-Carlo approach that is simpler than Bruegmann's (1993) approach. Our method is based on Abramson (1990). We performed experiments to assess ideas on (1) progressive pruning, (2) all moves as first heuristic, (3) temperature, (4) simulated annealing, and (5) depth-two tree search within the Monte-Carlo framework. Progressive pruning and the all moves as first heuristic are good speed-up enhancements that do not deteriorate the level of the program too much. Then, using a constant temperature is an adequate and simple heuristic that is about as good as simulated annealing. The depth-two heuristic gives deceptive results at the moment. The results of our Monte-Carlo programs against knowledge-based programs on 9x9 boards are promising. Finally, the ever-increasing power of computers lead us to think that Monte-Carlo approaches are worth considering for computer Go in the future.

Keywords: Monte-Carlo approach, computer Go, heuristics

1. Introduction

We start with two observations. First, when termination of the search process of a game tree is possible, the process provides the best move and constitutes a proof on that best move. The process does not necessarily need domain-dependent knowledge but its cost is exponential in the search depth. Second, a domain-dependent move generator generally yields a good move, but without any verification. It costs nothing in execution time but the move generator remains incomplete and always contains errors. When considering the game of Go, these two observations are crucial. Global tree search is not possible in Go and knowledge-based Go programs are very difficult to improve (Bouzy and Cazenave, 2001). Therefore, this paper explores an intermediate approach in

which a Go program performs a global search (not a global tree search) using very little knowledge. This approach is based on statistics or more specifically, on Monte-Carlo methods. We believe that such an approach does neither have the drawback of global tree search with very little domain-dependent knowledge (no termination), nor the drawback of domain-dependent move generation (no verification). The statistical global search described in this paper terminates and provides the move with a kind of verification. In this context, the paper claims the adequacy of statistical methods, or Monte-Carlo methods, to the game of Go.

To support our view, Section 2 describes related work about Monte-Carlo methods applied to Go. Section 3 focuses on the main ideas underlying our work. Then, Section 4 highlights the experiments to validate these ideas. Before conclusion, Section 5 discusses the relative merits of the statistical approach and its variants along with promising perspectives.

2. Related Work

At a practical level, the general meaning of Monte Carlo lies in the use of the random generator function, and for the theoretical level we refer to Fishman (1996). Monte-Carlo methods have already been used in computer games. In incomplete information games, such as poker (Billings et al., 2002), scrabble (Sheppard, 2002), and backgammon (Tesauro, 2002), this approach is natural: because the information possessed by your opponent is hidden, you want to simulate this information. In complete information games, the idea of replacing complete information by randomized information is less natural. Nevertheless, it is not the first time that Monte-Carlo methods have been tried in complete information games. This section deals with two previous contributions (Abramson, 1990; Bruegmann, 1993).

2.1 Abramson's Expected-Outcome

Evaluating a position of a two-person complete information game with statistics was tried by Abramson (1990). He proposed the expected-outcome model, in which the proper evaluation of a game-tree node is the expected value of the game's outcome given random play from that node on. The author showed that the expected outcome is a powerful heuristic. He concluded that the expected-outcome model of two-player games is "precise, accurate, easily estimable, efficiently calculable, and domain-independent". In 1990, he tried the expected-outcome model on the game of 6x6 Othello. The ever-increasing computer power enables us to use this model now for Go programs.

2.2 Bruegmann's Monte-Carlo Go

Bruegmann (1993) was the first to develop a Go program based on random games. The architecture of the program, GOBBLE, was remarkably simple. In

order to choose a move in a given position, GOBBLE played a large number of almost random games from this position to the end, and scored them. Then, he evaluated a move by computing the average of the scores of the random games in which it had been played.

This idea is the basis of our work. Below we describe some issues of GOBBLE. In our work, described in Section 3, they are subject to improvements.

1 *No filling of the eyes*. Moves that filled one's eyes were forbidden. This was the sole domain-dependent knowledge used in GOBBLE. In the game of Go, the groups must have at least two eyes in order to be alive (with the relatively rare exception of groups living in *seki*). If the eyes could be filled, the groups would never live and the random games would not actually finish. However, the exact definition of an eye has its importance.

2 *Evaluation of the moves*. Moves were evaluated according to the average score of the games in which they were played, not only at the beginning but *at any stage of the game*, provided that it was the first time one player had played at the intersection. This was justified by the fact that moves are often good independently of the stage at which they are played. However, this can turn out to be a fairly dangerous assumption.

3 *Selection of the moves*. Moves were not chosen completely randomly, but rather on their current evaluation, good moves having more chances to be played first. Moreover, simulated annealing was used to control the probability that a move could be played out of order. The amount of randomness put in the games was controlled by the *temperature*; it was set high at the beginning and gradually decreased. Thus, in the beginning, the games were almost completely random, and at the end they were almost completely determined by the evaluations of the moves. However, we will see that both are possible: (1) to fix the temperature to a constant value, and (2) to make the temperature even infinite, which means that all moves are played with equal probability.

3. Our Work

This section first describes the *basic idea* underlying our work (Subsection 3.1). Then, it presents our Go programs, OLGA and OLEG (Subsection 3.2). The only important domain-dependent consideration of the method, the definition of *eyes*, is described in Subsection 3.3. Finally, in Subsection 3.4 a graph explaining the various possible enhancements to the basic idea is given.

3.1 Basic Idea

Though the architecture of the GOBBLE program was particularly simple, some points were subject to discussion. Our own algorithm for Monte-Carlo Go programs is an adaptation of Abramson's (1990). The basic idea is: to evaluate

a position by playing a given number of completely random games to the end - without filling the eyes - and then scoring them. The evaluation corresponds to the mean of the scores of those random games. Choosing a move in a position means playing each of the moves and maximize the evaluations of the positions obtained at depth 1.

3.2 Two Programs: OLGA and OLEG

We developed two Go programs based on the basic idea above: OLGA and OLEG. OLGA and OLEG are far-fetched French acronyms for "ALeatoire GO" or "aLEatoire GO" that mean random Go. OLGA was developed by Bouzy (2002) as a continuation of the INDIGO development. The main idea was to use an approach with very little domain-dependent knowledge. At the beginning, the second idea in the OLGA development was to concentrate on the speed of the updating of the objects relative to the rules of the game, which was not highlighted in the previous developments of INDIGO. Of course, OLGA uses code available in INDIGO.

OLEG was written by Helmstetter. Here, the main idea was to reproduce the Monte-Carlo Go experiments of Bruegmann (1993) to obtain a Go program with very little Go knowledge. OLEG uses the basic data structure of GNUGO that is already very well optimized by the GNUGO team (Bump, 2003).

Both in OLEG and in OLGA, the quality of play depends on the precision expected that varies with the number of tests performed. The time to carry out these tests is proportional to the time spent to play one random game. On a 2 GHz computer, OLGA plays 7,000 random 9x9 games per second and OLEG 10,000.

Because strings, liberties, and intersection accessibilities are updated incrementally during the random games, the number of moves per second is almost constant and the time to play a game is proportional to the board size. Since the precision of the expected value depends on the square of the number of random games, there is no need to gain 20 per cent in speed, which would only bring about a 10-per-cent improvement in the precision. However, optimizing the program very roughly is important. A first pass of optimizations can gain a ratio of 10, and the precision can be three times better in such a case, which is worthwhile.

OLGA and OLEG share the basic idea and most of the enhancements that are described in Subsection 3.4. They are used to test the relative merits of each enhancement. However, each program uses its own definition of eyes.

3.3 How to Define Eyes?

The only domain-dependent knowledge required is the definition of an *eye*. It is important for the random program not to play a move in an eye. Without

this rule, the random player would never make living groups and the games would never end. There are different ways to define "eyes" as precisely as possible with domain-dependent knowledge such as Fotland (2002) and Chen and Chen (1999). Our definitions are designed to be integrated into a random Go-playing program; they are simple and fast but not correct in some cases.

In OLGA, an eye is an *empty intersection surrounded by stones of one colour with two liberties or more*.

In OLEG, an eye is an *empty intersection surrounded by stones belonging to the same string*.

The upside of both definitions is the speed of the programs. OLEG's definition is simpler and faster than OLGA's. Both approaches have the downside of being wrong in some cases. OLEG's definition is very restrictive: OLEG's eyes are actual true eyes but it may fill an actual eye surrounded by more than one string. Besides, OLGA has a fuzzy and optimistic definition: it never fills an actual eye but, to connect its stones surrounding an OLGA's eye, OLGA always expects one adjacent stone to be put into atari.

3.4 Various Possible Enhancements

So far, we have identified a few possible enhancements from the basic idea. They are shown in Figure 1. This figure also shows the enhancements used by OLEG and OLGA in their standard configurations. Two of the enhancements were already present in GOBBLE, namely the *all moves as first* heuristic (which means making statistics not only for the first move but for all moves of the random games) and *simulated annealing*. For the latter, an intermediate possibility can be adopted: instead of making the temperature vary during the game, we make it constant.

With a view of speeding up the basic idea, an alternative to the all-moves-as-first heuristic is *progressive pruning* of which only the first move of the random games is taken into account for the statistics, and the moves of which the evaluation is too low compared to the best move are pruned.

Making a minimax at depth 2 and evaluating the positions by making random games from this position naturally evolves from the basic idea. The expected result is an improvement of the program reading ability. For instance, it would suppress moves that work well only when the opponent does not respond.

4. Experiments

Starting from the basic idea, this section describes and evaluates the various enhancements: progressive pruning, all-moves-as-first heuristic, temperature, simulated annealing, and depth-two enhancements.

For each enhancement, we set up experiments to assess its effect on the level of our programs. One experiment consists in a match of 100 games between the

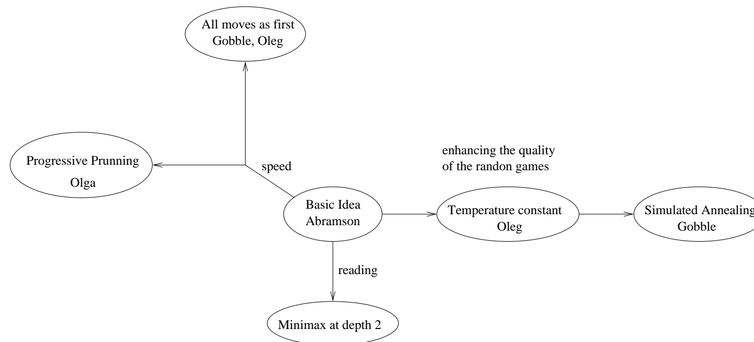

Figure 1. Possible enhancements.

program to be assessed and the experiment reference program, each program playing 50 games with Black. In most experiments, the program to be assessed is a program in which one parameter varies, and the reference program is the same program with the parameter fixed to a reference value. In the other set of experiments, the program to be assessed uses the enhancement while the reference program does not. The result of an experiment is generally a set of relative scores provided by a table assuming that the program of the column is the max player. Given that the standard deviation of 9x9 games played by our programs is roughly 15 points, 100 games enable our experiments to lower σ down to 1.5 points and to obtain a 95% confidence interval of which the radius equals 2σ, i.e., 3 points. We have used 2 GHz computers. When the response time of the assessed program varies with the experimental parameters, we mention it. Furthermore, all programs in this work do not use any conservative or aggressive style depending on who is ahead in a game, they only try to maximize their own score. The score of a game is more significant than the winning percentage which is consequently not included in the experiments' results. We terminate this section with an assessment of OLGA and OLEG against two existing knowledge-based programs INDIGO and GNUGO, in showing the results of an all-against-all tournament.

4.1 Progressive Pruning

As contained in the basic idea, each move has a mean value m, a standard deviation σ, a left expected outcome m_l and a right expected outcome m_r. For a move, $m_l = m$ - σr_d and $m_r = m + \sigma r_d$. r_d is a ratio fixed up by practical experiments. A move M_1 is said to be statistically inferior to another move M_2 if $M_1.m_r < M_2.m_l$. Two moves M_1 and M_2 are statistically equal when $M_1.\sigma < \sigma_e$ and $M_2.\sigma < \sigma_e$ and no move is statistically inferior to the other. σ_e is called standard deviation for equality, and its value is determined by experiments.

In *Progressive Pruning* (PP), after a minimal number of random games (100 per move), a move is pruned as soon as it is statistically inferior to another move. Therefore, the number of candidate moves decreases while the process is running. The process stops either when there is only one move left (this move is selected), or when the moves left are statistically equal, or when a maximal threshold of iterations is reached. In these two cases, the move with the highest expected outcome is chosen. The maximal threshold is fixed to 10,000 multiplied by the number of legal moves. This progressive pruning algorithm is similar to the one described in Billings et al. (2002).

Due to the increasing precision of mean evaluations while the process is running, the max value of the root is decreasing. Consequently, a move can be statistically inferior to the best one at a given time and not later. Thus, the pruning process can be either hard (a pruned move cannot be a candidate later on) or soft (a move pruned at a given time can be a candidate later on). Of course, soft PP is more precise than hard PP. Nevertheless, in the experiments shown here, OLGA uses hard PP.

r_d	1	2	4	8
mean	0	+5.6	+7.3	+9.0
time	10'	35'	90'	150'

Table 1. Times and relative scores of PP with different values of r_d, against PP(r_d=1).

σ_e	0.2	0.5	1
mean	0	-0.7	-6.7
time	10'	9'	7'

Table 2. Times and relative scores of PP with different values of σ_e, against PP(σ_e=0.2).

The inferiority of one move compared to another, used for pruning, depends on the value of r_d. Theoretically, the greater r_d is, the less pruned the moves are, and, as a consequence, the better the algorithm performs, but the slower it plays. The equality of moves, used to stop the algorithm, is conditioned by σ_e. Theoretically, the smaller σ_e is, the fewer equalities there are, and the better the algorithm plays but with an increased slowness. We set up experiments with different versions of OLGA to obtain the best compromise between the time and the level of the program. The first set of experiments consisted in assessing the level and speed of OLGA depending on r_d. OLGA(r_d) played a set of games either with black or white against OLGA(r_d=1). Table 1 shows the mean of the relative score of OLGA(r_d) when r_d varies from 1 up to 8. Both the minimal number of random games and the maximal threshold remain constant (100 and 10,000 respectively).

This experiment shows that r_d plays an important role in the move pruning process. Large values of r_d correspond to the basic idea. To sum up, progressive pruning loses little strength compared to the basic idea, between five or ten points according to the value of r_d. In the next experiments, r_d is set to 1. The second set of experiments deals with σ_e in the same way. Table 2 shows the mean of the relative score of OLGA(σ_e) when σ_e varies from 0.2 up to 1.

OLGA(σ_e=1) yields the worst score while using less time. This experiment confirms the role played by σ_e in the move pruning process. In the next experiments, σ_e is set to 0.2.

4.2 The All-Moves-As-First Heuristic

When evaluating the terminal position of a given random game, this terminal position may be the terminal position of many other random games in which the first move and another friendly move of the random game are reversed. Therefore, when playing and scoring a random game, we may use the result either for the first move of the game only, or *for all moves played in the game as if they were the first to be played*. The former is the basic idea, the latter is what was performed in GOBBLE, and we use the term *all moves as first* heuristic.

4.2.1 Advantages and Drawbacks. The idea is attractive, because one random game helps evaluate almost all possible moves at the root. However, it does have some drawbacks because the evaluation of a move from a random game in which it was played at a late stage is less reliable than when it is played at an early stage. This phenomenon happens when captures have already occurred at the time when the move is played. In figure 2 the values of the moves *A* for Black and *B* for White largely depend on the order in which they are played.

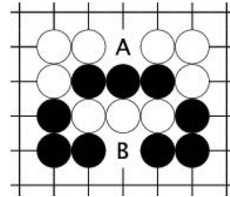

Figure 2. The move order is important.

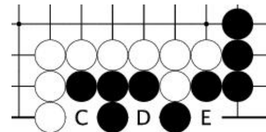

Figure 3. The value of moves may be very different for both players.

There might be more efficient ways to analyse a random game and decide whether the value of a move is the same as if it was played at the root. Thus, we would obtain the best of both worlds: efficiency and reliability. To this end, at least one easy thing should be done (it has already been done in GOBBLE and in OLEG): in a random game, if several moves are played at the same place because of captures, modify the statistics only for the player who played first.

The method has another troublesome side-effect: it does not evaluate the value of an intersection for the player to move but rather the difference between the values of the intersection when it is played by each player. Indeed, in most random games, any intersection will be played either by one player or the other, with an equal probability of about $1/2$ (an intersection is almost always played at least once during a random game). Therefore, the average score of all random games lies approximately in the middle between the average score when White has played a move and the average score when Black has played a move. Most often, this problem is not serious, because the value of a move for one player

is often the same for both players; but sometimes it is the opposite. In Figure 3 the point *C* is good for White and bad for Black. On the contrary *D* and *E* are good for Black only.

4.2.2 Experimental Comparison with Progressive Pruning. Compared to the very slow basic idea the gain in speed offered by the all-moves-as-first heuristic is very important. In contrast to the basic idea or PP, the number of random games to be played becomes independent of the number of legal moves. This is the main feature of this heuristic. Instead of playing a 9x9 game in more than two hours by using the basic idea, OLGA plays in five minutes with the use of this heuristic. However, we have seen two problems due to the use of this heuristic. Therefore, how do the uses of all moves as first heuristic and progressive pruning compare in strength?

Table 3 shows the mean of the relative scores of OLGA(Basic idea) and OLGA(PP) against OLGA(all moves as first).

Basic idea	PP
+13.7	+4.0

Table 3. Relative scores of OLGA with the basic idea or with PP, against the all-moves-as-first heuristic.

While the previous section underlines that PP decreases the level of OLGA by about five or ten points according to the value of r_d, the all-moves-as-first heuristic decreases the level by almost fifteen points. The confrontation between OLGA(PP) and OLGA(all moves as first) shows that PP remains better in strength.

4.2.3 Influence of the Number of Random Games. The standard deviation σ of the random games usually amounts to 45 points at the beginning and in the middle game, and diminishes in the endgame. If we play N random games and take the average, the standard deviation is $\sigma/\sqrt{N}$. This calculation helps find how many random games to play so that the evaluations of the moves become sufficiently close to their expected outcome. From a practical point of view the question is: how does this relate to the level of play? Table 4 shows the result of Oleg($N = 10,000$) against OLEG($N = 1000$) and OLEG($N = 100,000$).

1000	100,000
-12.7	+3.2

Table 4. Relative scores of OLEG with different values of N, against OLEG($N = 10,000$).

We can conclude that 10,000 random games per move is a good compromise when using the all-moves-as-first heuristic. Since Oleg is able to play 10,000 random games per second, this means it can play one move per second while using only this heuristic.

4.3 Temperature

Instead of making the temperature start high and decrease as we play more random games, it is simpler to make it a constant. The temperature has been

implemented in OLEG in a somewhat different way as in GOBBLE. In the latter, two lists of moves were maintained for both players, and the moves in the random games were played in the order of the lists (if the move in the list is not legal, we just take the next in the list). Between each random game, the lists were sorted according to the current evaluation of the moves and then moves were shifted in the list with a probability depending on the temperature.

In OLEG, in order to choose a move in a random game, we consider all the legal moves and play one of them with a probability proportional to

$$\exp(Kv),$$

where v is the current evaluation of the move and K a constant which must be seen as the inverse of the temperature ($K = 0$ means $T = \infty$). A drawback of this method is that it slows down the speed of the random games to about 2,000 per second. Table 5 shows the results of OLEG($K = 2$) against OLEG(K) for a few values of K.

K	0	5	10	20
mean	-8.1	+2.6	-4.9	-11.3

Table 5. Relative scores of OLEG with different values of K against OLEG(K=2).

So, there is indeed something to be gained by using a constant temperature. This is probably because the best moves are played early and thus, obtain a more accurate evaluation. However, it is bad to have K too large. The best we have found is $K = 5$.

4.4 Simulated Annealing

Simulated annealing (Kirkpatrick, Gelatt, and Vecchi, 1983) was presented in Bruegmann (1993) as the main idea of the method. We have seen that it is perfectly possible not to use it, so the question arises: what is its real contribution?

To answer the question we performed some experiments with simulated annealing in OLEG. In our implementation the variable K increases as more random games are played. However, we have not been able to achieve significantly better results this way than with K set to a constant. For example, we have made an experiment between OLEG with simulated annealing and K varying from 0 to 5, and OLEG with $K = 5$. The version with simulated annealing won by 1.6 points in average.

The motivation for using simulated annealing was probably that the program would gain some reading ability, but we have not seen any evidence of this, the program making the same kind of tactical blunders. Besides, the way simulated annealing is implemented in GOBBLE is not classical. Simulated annealing normally has an evaluation that depends only on the current state (in the case of GOBBLE, a state is the lists of moves for both players); instead in GOBBLE the evaluation of a state is the average of all the random games that are based on all the states reached so far. There may be a way to design a

true simulated-annealing-based Go program, but speed would, then, be a major concern.

4.4.1 OLEG against VEGOS. VEGOS is a recent go program available on the web (Kaminski, 2003). It is based on the same ideas as GOBBLE; particularly it uses simulated annealing. A confrontation of 20 games against OLEG($K = 2$, without simulated annealing) has resulted in an average win of 7.5 points for OLEG. We did not perform more games because we had to play them by hand. The playing styles of the programs are similar, with slightly different tactical weaknesses. The result of this confrontation is another reason why we doubt that simulated annealing is crucial for Monte-Carlo Go.

4.5 Depth-2 Enhancement

For the depth-2 enhancement the given position is the root of a depth-two min-max tree. Let us start the random games from the root by two given moves, one move for the friendly side, and, then, one move for the opponent, and make statistics on the terminal position evaluation for each node situated at depth 2 in the min-max tree. At depth-one nodes, the value is computed by using the min rule. When a depth-one value has been proved to be inferior to another one, then this move is pruned, and no more random games are started with this move first. This variant is more complex in time because, if n is the number of possible moves, about n^2 statistical variables must be sampled, instead of n only.

We set up a match between two versions of OLGA using progressive pruning at the root node. OLGA($Depth$=1) backs up the statistics about random games at depth one while OLGA($Depth$=2) backs up the statistics at depth two and uses the min rule to obtain the value of depth-one nodes. The values of the parameters of OLGA($Depth$=1) are the same as the parameters of the PP program. The minimal number of random games without pruning is set to 100. The maximal number of random games is also fixed to 10,000 multiplied by the number of legal moves, r_d is set to 1, and σ_e is set to 0.2. While OLGA($Depth$=1) only uses 10' per 9x9 game, OLGA($Depth$=2) is very slow. In order to speed up OLGA($Depth$=2), we use the all moves as first heuristic. Thus, it uses about 2 hours per 9x9 game, which yields results in a reasonable time.

Table 6 shows the mean of the relative score of Prog($Depth$=2) against Prog($Depth$=1), Prog being either OLGA or OLEG.

OLGA	OLEG
-2.1	-2.4

Table 6. Relative scores of Prog($Depth$=2) against Prog($Depth$=1).

Intuitively, the results should be better for the depth-two programs, but they are actually slightly worse. How can this be explained?

The first possible explanation lies in the min-max oscillation observed at the root node when performing iterative deepening. A depth-one search overestimates

the min-max value of the root while a depth-two search underestimates the min-max value. Thus, the depth-two min-max value of the root node is more difficult to separate from the evaluation of the root (also obtained with random simulations) than the depth-one min-max value is. In this view, OLGA($Depth$=2) pass on some positions on which OLGA($Depth$=1) does not. In order to obtain an answer to the validity of this explanation, a depth-three experiment becomes mandatory. If depth three performs well, then the explanation should be reinforced, otherwise another explanation is needed.

The second explanation is statistical. Let Z be a random variable which is the maximum of 10 identical random variables X_i ($0 \leq i \leq 9$) with mean(X_i) = 0 and standard deviation $\sigma(X_i) = 1$, plus a last one Y with mean(Y) = $\delta > 0$ and standard deviation $\sigma(Y) = 1$. We have Z = max(X_0, ..., X_9, Y). Table 7 provides the mean and standard deviation of Z.

δ	0	1	2	3	4
mean(Z)	1.58	1.77	2.27	3.06	4.01
$\sigma(Z)$	0.58	0.62	0.77	0.92	0.98

Table 7. Mean and standard deviation of Z with different values of δ.

Table 7 shows that, on positions in which all 11 moves are equals ($\delta = 0$), performing a max (resp. min) leads to a positive (resp. negative) value (1.58) significantly greater (resp. smaller) than the (resp. opposite of the) standard deviation of each move (1). Therefore, when performing a depth-two search, the depth-one nodes are largely underestimated and, given these depth-one estimations, the root node is largely overestimated. Thus, when the number of games is not sufficient, the error propagates once in the negative direction and then in the positive one. To sum up, when the moves are almost equal, the min-max value at the root node contains a great deal of randomness.

Table 7 also points out another explanation. When $\delta \leq 2$, mean(Z) and $\sigma(Z)$ remain quite different from δ and 1 respectively. But when $\delta \geq 4$, both mean(Z) and $\sigma(Z)$ are almost equal to δ and 1 respectively. Thus, on positions with one best move only and ten average moves, the mean value of the max value becomes exact only when the difference between the best move evaluation and the other move evaluation is about four times the value of the standard deviation of the move evaluations.

These two remarks show that, when using the depth-two enhancement, a great deal of uncertainty is contained in the min value of depth-one nodes and even more in the min-max value of the root node.

4.6 An All-against-All Tournament

To evaluate the Monte-Carlo approach against the knowledge-based approach, this subsection provides the results of an all-against-all 9x9 tournament between OLGA, OLEG, INDIGO and GNUGO. GNUGO (Bump, 2003)

is a knowledge-based Go program developed by the Free Software Foundation. We used the 3.2 version released in April 2002. INDIGO2002 (Bouzy, 2002) is another knowledge-based program whose move decision process is described in Bouzy (2003). OLGA means OLGA($Depth$=1, r_d=1, σ_e=0.2) using PP and not the all-moves-as-first heuristic. OLEG uses the all-moves-as-first heuristic, a constant temperature corresponding to K=2, and it does not use PP. Table 8 shows the grid of the all against all tournament.

	Olga	Indigo	GnuGo
Oleg	+10.4	-4.9	+31.5
Olga		+1.8	+33.7
Indigo			+8.7

Table 8. The grid of the all against all tournament.

First, Monte Carlo excepted, our tests show that, on 9x9 board, GNUGO 3.2 is about 8.7 points better than INDIGO2002. Then, considering Monte Carlo, both OLGA and OLEG are far below GNUGO (more than thirty points average). However, given the very large difference of complexity between the move generator of GNUGO and our move generators, this result is quite satisfactory. Against INDIGO, both OLGA and OLEG perform well. The three programs beat themselves circularly. On 9x9 boards, we may say that OLEG and OLGA containing very little knowledge have a comparable level to the level of INDIGO that contains a large amount of knowledge. The result between two very different architectures, statistical and knowledge, is quite enlightening.

Besides, we have made tests on larger boards. Although the number of games played is not sufficient to obtain significant results, they give an idea of the behaviour of Monte-Carlo programs in such situations. On the basis of twenty 13x13 games only, OLGA is 17 points below INDIGO. On a 19x19 Go board, a 7 games' confrontation between OLEG and GNUGO was won by GNUGO with an average margin of 83 points. OLEG takes a long time to play (about 3 hours per game) for several reasons. First, the random games are longer. Second, we must play more of them to have an accurate evaluation of the moves (we did it with 50,000 random games per move). Lastly, the main game itself is longer. In those games, typically OLEG makes a large connected group in the centre with just sufficient territory to live and GNUGO gets the points on the sides.

5. Discussion

While showing a sample game between OLEG and its author, this section discusses the strengths and weaknesses of the statistical approach and opens up some promising perspectives.

5.1 Strengths and Weaknesses

On the programmer's side, the main strength of the Monte-Carlo approach is that it uses very little knowledge. First, a Monte-Carlo game program can

be developed very quickly. As Bruegmann (1993) did for the game of Go, this upside must be underlined: the programmer has to implement efficiently the rules of the game and eyes, and that is all. He can leave all other knowledge aside. Second, the decomposition of the whole game into sub-games, a feature of knowledge-based programs, is avoided. This decomposition introduces a bias in knowledge-based programs, and Monte-Carlo programs do not suffer from this downside. Finally, the evaluations are performed on terminated games, and, consequently, the evaluation function is trivial. Besides, Monte-Carlo Go programs are weak tactically, and they are still slower than classical programs and, at the moment, it is difficult to make them play on boards larger than 13x13.

In the human user's viewpoint, any Monte-Carlo Go program underestimates the positions for both sides. Thus, it likes to keep its own strength. As a result, it likes to make strongly connected shapes. Conversely, it looks for weaknesses in the opponent position that do not exist. This can be seen in the game of Figure 4. It was played between OLEG as Black and its author as White. OLEG was set with $K = 5$ and 10,000 random games per move. White was playing relatively softly in this game and did not try to crush the program.

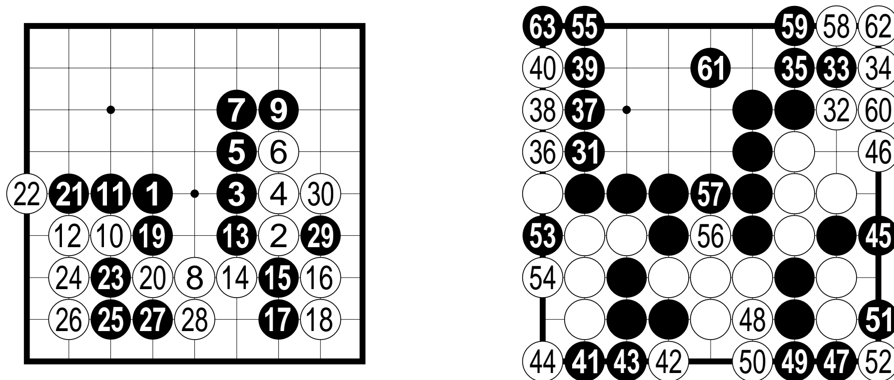

Figure 4. OLEG(B)-Helmstetter(W). White wins by 17 points plus the komi.

5.2 Perspectives

First, eliminate the tactical weakness of the Monte-Carlo method with a processing containing tactical search. Second, use domain dependent knowledge to play pseudo-random games. Third, build statistics not only on the global score but on other objects.

5.2.1 Preprocessing with Tactical Search. The main weakness of the Monte-Carlo approach is tactics. Therefore, it is worth adding some tactical modules to the program. As a first step it is easy to add a simple tactical module which reads ladders. This module can be either a preprocessing module

or a post-processing module to the Monte-Carlo method. In this context, each module is independent of the other one, and does not use the strength of the other one. Another idea would consist in making the two modules interact. When the tactical module selects moves for the random games, it would be useful for Monte Carlo to use the already available tactical results. This approach would require a quick access to the tactical results, and would slow down the random games. The validity of the tactical results would depend on the moves already played and it would be difficult to build an accurate mechanism to this end. Nevertheless, this approach looks promising.

5.2.2 Using Domain Dependent Pseudo-random Games. Until now, a program using random games and very little knowledge has a level comparable to INDIGO2002. Thus, what would be the level of a program using domain dependent pseudo-random games? As suggested by Bruegmann (1993), a first experiment would be to make the random program use patterns giving the probability of a move advised by the pattern. The pattern database should be built a priori and should not introduce too much bias into the random games.

5.2.3 Exploring the Locality of Go with Statistics. To date, we have estimated the value of a move by considering only the final scores of the random games where it had been played. Thus, we obtain a global evaluation of the move. This is both a strength and a weakness of the method. Indeed, the effect of a move is often only local, particularly on 19x19 go boards. We would like to know whether and why a move is good.

It might be possible to link the value of a move to more local subgoals from which we could establish statistics. The value of those subgoals could, then, be evaluated by linking them to the final score. Interesting subgoals could deal with capturing strings or connecting strings together.

6. Conclusion

In this paper, we described a Monte-Carlo approach to computer Go. Like Bruegmann's (1993) Monte-Carlo Go, it uses very little domain-dependent knowledge, except concerning eyes. When compared to the knowledge-based approaches, this approach is very easy to implement. However, its weakness lies in the tactics. We have assessed several heuristics by performing experiments with different versions of our programs OLGA and OLEG. Progressive pruning and the all-moves-as-first heuristic enables the programs to play more quickly without decreasing their level much. Then, adding a constant temperature to the approach guarantees a higher level but yields a slightly slower program. Furthermore, we have shown that adding simulated annealing does not help: it makes the program more complicated and slower, and the level is not significantly better. Besides, we have tried to enhance our programs with

a depth-two tree search, which did not work well. Lastly, we have assessed our programs against existing knowledge-based ones, GnuGo and Indigo, on 9x9 boards. Olga and Oleg are still clearly inferior to GnuGo (version 3.2) but they match Indigo.

We believe that, with the help of the ever-increasing power of computers, this approach is promising for computer Go in the future. At least, it provides Go programs with a statistical global search, which is less expensive than global tree search, and which enriches move generation with a kind of verification. In this respect, this approach fills the gap left by global tree search in computer Go (no termination) and left by move generation (no verification). We believe that the statistical search is an alternative to tree search (Junghanns, 1998) worth considering in practice. It has already been considered theoretically within the framework of Rivest (1988). In the near future, we plan to enhance our Monte-Carlo approach in several ways: adding tactics, inserting domain-dependent knowledge into the random games, and exploring the locality of Go with more statistics.

References

Abramson, B. (1990). Expected-outcome : a general model of static evaluation. *IEEE transactions on PAMI*, Vol. 12, pp. 182–193.

Billings, D., Davidson, A., Schaeffer, J., and Szafron, D. (2002). The challenge of poker. *Artificial Intelligence*, Vol. 134, pp. 201–240.

Bouzy, B. (2002). Indigo home page. http://www.math-info.univ-paris5.fr/˜bouzy/INDIGO.html.

Bouzy, B. (2003). The move decision process of Indigo. *ICGA Journal*, Vol. 26, No. 1, pp. 14–27.

Bouzy, B. and Cazenave, T. (2001). Computer Go: an AI oriented survey. *Artificial Intelligence*, Vol. 132, pp. 39–103.

Bruegmann, B. (1993). Monte Carlo Go.
ftp://www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z.

Bump, D. (2003). Gnugo home page. http://www.gnu.org/software/gnugo/devel.html.

Chen, K. and Chen, Z. (1999). Static analysis of life and death in the game of Go. *Information Sciences*, Vol. 121, Nos. 1-2, pp. 113–134.

Fishman (1996). *Monte-Carlo : Concepts, Algorithms, Applications*. Springer-Verlag, Berlin, Germany.

Fotland, D. (2002). Static Eye in "The Many Faces of Go". *ICGA Journal*, Vol. 25, No. 4, pp. 203–210.

Junghanns, A. (1998). Are there Practical Alternatives to Alpha-Beta? *ICCA Journal*, Vol. 21, No. 1, pp. 14–32.

Kaminski, P. (2003). Vegos home page. http://www.ideanest.com/vegos/.

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. (1983). Optimization by Simulated Annealing. *Science*.

Rivest, R. (1988). Game-tree searching by min-max approximation. *Artificial Intelligence*, Vol. 34, No. 1, pp. 77–96.

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

Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, Vol. 134, Nos. 1-2, pp. 181–199.